Layered Software Architecture
<table>
<thead>
<tr>
<th>Document Title</th>
<th>Layered Software Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Owner</td>
<td>AUTOSAR</td>
</tr>
<tr>
<td>Document Responsibility</td>
<td>AUTOSAR</td>
</tr>
<tr>
<td>Document Identification No</td>
<td>053</td>
</tr>
<tr>
<td>Document Classification</td>
<td>Auxiliary</td>
</tr>
<tr>
<td>Document Version</td>
<td>3.2.0</td>
</tr>
<tr>
<td>Document Status</td>
<td>Final</td>
</tr>
<tr>
<td>Part of Release</td>
<td>4.0</td>
</tr>
<tr>
<td>Revision</td>
<td>3</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| 06.10.2011 | 3.2.0   | AUTOSAR Administration   | - added a note for the R3-compatibility FlexRay Transport Layer FrArTp on slide "ki890".  
- added an overview chapter for energy management and partial networking  
- corrected examples regarding DEM symbol generation  
- fixed minor typography issues  
- clarification of term AUTOSAR-ECU on slide "94jt1"  
- corrected CDD access description for EcuM on slide "11123" |
| 24.11.2010 | 3.1.0   | AUTOSAR Administration   | - added a note regarding support for System Basis Chips on slide "94juq"  
- clarification of DBG and DLT text on slide "3edfg"  
- corrected DBG description on slide "11231" |
| 30.11.2009 | 3.0.0   | AUTOSAR Administration   | - The document has been newly structured. There are now 3 main parts:  
  - Architecture  
  - Configuration  
  - Integration and Runtime aspects  
- The whole content has been updated to reflect the content of the R 4.0 specifications.  
- Topics which have bee newly introduced or heavily extended in release 4.0 have been added. E.g., Multi Core systems, Partitioning, Mode Management, Error Handling, Reporting and Diagnostic, Debugging, Measurement and Calibration, Functional Safety etc  
- Legal disclaimer revised |
<p>| 23.06.2008 | 2.2.1   | AUTOSAR Administration   | - Legal disclaimer revised |</p>
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changed by</th>
<th>Change Description</th>
</tr>
</thead>
</table>
| 15.11.2007 | 2.2.0   | AUTOSAR Administration | ✚ Updates based on new wakeup/startup concepts  
                |                      | ✚ Detailed explanation for post-build time configuration  
                |                      | ✚ "Slimming" of LIN stack description  
                |                      | ✚ ICC2 figure  
                |                      | ✚ Document meta information extended  
                |                      | ✚ Small layout adaptations made |
| 06.02.2007 | 2.1.0   | AUTOSAR Administration | ✚ ICC clustering added.  
                |                      | ✚ Document contents harmonized  
                |                      | ✚ Legal disclaimer revised  
                |                      | ✚ Release Notes added  
                |                      | ✚ "Advice for users" revised  
                |                      | ✚ "Revision Information" added |
| 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

**Disclaimer**

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

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

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

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

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

**Advice for users**

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

## 1. Architecture
   1. **Overview of Software Layers**
   2. Content of Software Layers
   3. Content of Software Layers in Multi Core systems
   4. Overview of Modules
   5. Interfaces
      1. General
      2. Interaction of Layers (Examples)

## 2. Configuration

## 3. Integration and Runtime aspects
Introduction
Purpose and Inputs

Purpose of this document
The Layered Software Architecture describes the software architecture of AUTOSAR:
- it describes in a top-down approach the hierarchical structure of AUTOSAR software and
- maps the Basic Software Modules to software layers and
- shows their relationship.

This document does not contain requirements and is informative only. The examples given are not meant to be complete in all respects.

This document focuses on static views of a conceptual layered software architecture:
- it does not specify a structural software architecture (design) with detailed static and dynamic interface descriptions,
  - these information are included in the specifications of the basic software modules themselves.

Inputs
This document is based on specification and requirement documents of AUTOSAR.
Introduction
Scope and Extensibility

Application scope of AUTOSAR
AUTOSAR is dedicated for Automotive ECUs. Such ECUs have the following properties:

- strong interaction with hardware (sensors and actuators),
- connection to vehicle networks like CAN, LIN, FlexRay or Ethernet,
- microcontrollers (typically 16 or 32 bit) with limited resources of computing power and memory (compared with enterprise solutions),
- Real Time System and
- program execution from internal or external flash memory.

NOTE: In the AUTOSAR sense an ECU means one microcontroller plus peripherals and the according software/configuration. The mechanical design is not in the scope of AUTOSAR. This means that if more than one microcontroller in arranged in a housing, then each microcontroller requires its own description of an AUTOSAR-ECU instance.

AUTOSAR extensibility
The AUTOSAR Software Architecture is a generic approach:

- standard modules can be extended in functionality, while still being compliant,
  - still, their configuration has to be considered in the automatic Basic SW configuration process!
- non-standard modules can be integrated into AUTOSAR-based systems as Complex Drivers and
- further layers cannot be added.
The AUTOSAR Architecture distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller.
The AUTOSAR Basic Software is further divided in the layers: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers.
The Basic Software Layers are further divided into functional groups. Examples of Services are System, Memory and Communication Services.
The **Microcontroller Abstraction Layer** is the lowest software layer of the Basic Software. It contains internal drivers, which are software modules with direct access to the μC and internal peripherals.

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

**Properties**
Implementation: μC dependent
Upper Interface: standardized 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
The Complex Drivers Layer spans from the hardware to the RTE.

**Task**
Provide the possibility to integrate special purpose functionality, e.g. drivers for devices:
- which are not specified within AUTOSAR,
- with very high timing constraints or
- for migration purposes etc.

**Properties**
Implementation: might be application, μC and ECU hardware dependent
Upper Interface: might be application, μC and ECU hardware dependent
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, error memory and fault treatment)
- ECU state management, mode management
- Logical and temporal program flow monitoring (Wdg manager)

**Task**

Provide basic services for applications and basic software modules.

**Properties**

Implementation: mostly µC and ECU hardware independent

Upper Interface: µC and ECU hardware independent
The **RTE** is a layer providing communication services to 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)**
  Standardized access to sensors, actuators and ECU onboard peripherals

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

- **Communication**
  Standardized access to: vehicle network systems, ECU onboard communication systems and ECU internal SW

- **System**
  Provision of standardizeable (operating system, timers, error memory) and ECU specific (ECU state management, watchdog manager) services and library functions
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 driver for an internal device is called **internal driver** and is located in the Microcontroller Abstraction Layer.
**Driver (external)**

External devices are located on the ECU hardware outside the microcontroller. Examples for external devices are:

- External EEPROM
- External watchdog
- External flash

A 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. This way also components integrated in System Basis Chips (SBCs) like transceivers and watchdogs are supported by AUTOSAR.

- **Example:** a driver for an external EEPROM with SPI interface accesses the external EEPROM via the handler/driver for the SPI bus.

**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.
An **Interface (interface module)** contains the functionality to abstract from modules which are architecturally placed below them. E.g., an interface module which abstracts from the hardware realization of a specific device. 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 and independent of the hardware realization (on chip, off chip).
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.

Handler functionality is often incorporated in the driver or interface (e.g. SPIHandlerDriver, ADC Driver).
A manager offers specific services for multiple clients. It is needed in all cases where pure handler functionality is not enough to abstract from multiple clients.

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 distributed and reliable data storage, data checking, provision of default values etc.
**Libraries** are a collection of functions for related purposes

Libraries:
- can be called by BSW modules (that including the RTE), SW-Cs, libraries or integration code
- run in the context of the caller in the same protection environment
- can only call libraries
- are re-entrant
- do not have internal states
- do not require any initialization
- are synchronous, i.e. they do not have wait points

The following libraries are specified within AUTOSAR:
- Fixed point mathematical,
- Floating point mathematical,
- Interpolation for fixed point data,
- Interpolation for floating point data,
- Bit handling,
- E2E communication,
- CRC calculation,
- Extended functions (e.g. 64bits calculation, filtering, etc.) and
- Crypto
1. **Architecture**
   1. Overview of Software Layers
   2. **Content of Software Layers**
   3. Content of Software Layers in Multi Core systems
   4. Overview of Modules
   5. Interfaces
      1. General
      2. Interaction of Layers (Examples)

2. **Configuration**

3. Integration and Runtime aspects
Architecture – Content of Software Layers
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)
The **SPIHandlerDriver** allows concurrent access of several clients to one or more SPI busses.

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

**Example:**

![Diagram showing the relationship between various drivers and abstraction layers.](image-url)
A **Complex Driver** is a module which implements non-standardized functionality within the basic software stack.

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

- Injection control
- Electric valve control
- Incremental position detection

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

**Properties:**
Implementation: highly µC, ECU and application dependent

Upper Interface to SW-Cs: specified and implemented according to AUTOSAR (AUTOSAR interface)

Lower interface: restricted access to Standardized Interfaces

---

**Example:**

- Complex Drivers
- Incremental Position Detection
- Electric Valve Control
- Injection Control

---

- **AUTOSAR Confidential** -

---

**Table:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller Drivers</td>
<td>Memory Drivers</td>
<td>Communication Drivers</td>
<td>I/O Drivers</td>
<td></td>
</tr>
</tbody>
</table>
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 might be 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)
**Architecture – Content of Software Layers**

**ECU Abstraction: Communication Hardware Abstraction**

The **Communication Hardware Abstraction** is a group of modules which abstracts from the **location** of communication controllers and the **ECU hardware layout**. For all communication systems a **specific** Communication Hardware Abstraction is required (e.g. for LIN, CAN, 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 its 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

---

### Table: System Services

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Services</td>
<td></td>
</tr>
<tr>
<td>Memory Services</td>
<td></td>
</tr>
<tr>
<td>Communication Services</td>
<td></td>
</tr>
<tr>
<td>I/O HW Abstraction</td>
<td></td>
</tr>
</tbody>
</table>

### Diagram: Communication Hardware Abstraction

- **CAN Interface**
- **CAN Transceiver Driver**
- **Driver for ext. CAN ASIC**
- **I/O Drivers**
- **Communication Drivers**
- **µC**
- **SPI**
- **CAN**

- **AUTOSAR Confidential**
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 are accessible via the same mechanism.

The memory drivers are accessed via memory specific abstraction/emulation modules (e.g. EEPROM Abstraction).

By emulating an EEPROM abstraction on top of 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
The **Onboard Device Abstraction** contains drivers for ECU onboard devices which cannot be seen as sensors or actuators like internal or external watchdogs. 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
The **Communication Services** are a group of modules for vehicle network communication (CAN, LIN and FlexRay). They interface with the communication drivers via the communication hardware abstraction.

**Task:**
- Provide a uniform interface to the vehicle network for communication.
- Provide uniform services for network management
- Provide uniform interface to the vehicle network for diagnostic communication
- 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.
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.

**Please Note:**
- There are two transport protocol modules in the CAN stack which can be used alternatively or in parallel: CanTp and J1939Tp. They are used as follows:
  - CanTp: ISO Diagnostics (DCM), large PDU transport on standard CAN bus
  - J1939Tp: J1939 Diagnostics, large PDU transport on J1939 driven CAN bus
Properties:

- Implementation: µC and ECU HW independent, partly dependent on CAN.
- AUTOSAR COM, Generic NM (Network Management) Interface and Diagnostic Communication Manager are the same for all vehicle network systems and exist as one instance per ECU.
- Generic NM Interface contains only a dispatcher. No further functionality is included. In case of gateway ECUs it can also include the NM coordinator functionality which allows to synchronize multiple different networks (of the same or different types) to synchronously wake them up or shut them down.
- CAN Generic NM is specific for CAN networks and will be instantiated per CAN vehicle network system. CAN Generic NM interfaces with CAN via underlying network adapter (CAN NM).
- The communication system specific Can State Manager handles the communication system dependent Start-up and Shutdown features. Furthermore it controls the different options of COM to send PDUs and to monitor signal timeouts.
The **TTCAN Communication Services** are the optional extensions of the plain CAN Interface and CAN Driver module for vehicle network communication with the communication system TTCAN.

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

**Please Note:**
- The CAN Interface with TTCAN can serve both a plain CAN Driver and a CAN Driver TTCAN.
Properties:

- TTCAN is an absolute superset to CAN, i.e. a CAN stack which supports TTCAN can serve both a CAN and a TTCAN bus.
- CanIf and CanDrv are the only modules which need extensions to serve TTCAN communication.
- The properties of the communication stack CAN are also true for CAN with TTCAN functionality.
The **LIN Communication Services** are a group of modules for vehicle network communication with the communication system LIN.

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

**Properties:**

The **LIN Communication Services** contain:
- A LIN 2.1 compliant communication stack with
  - Schedule table manager for transmitting LIN frames and to handle requests to switch to other schedule tables.
  - Transport protocol, used for diagnostics
  - A WakeUp and Sleep Interface
- An underlying LIN Driver:
  - implementing the LIN protocol and adaptation the specific hardware
  - Supporting both simple UART and complex frame based LIN hardware
Note: Integration of LIN into AUTOSAR:

- Lin Interface controls the WakeUp/Sleep API and allows the slaves to keep the bus awake (decentralized approach).
- The communication system specific LIN State Manager handles the communication dependent Start-up and Shutdown features. Furthermore it controls the communication mode requests from the Communication Manager. The LIN state manager also controls the I-PDU groups by interfacing COM.
- When sending a LIN frame, the LIN Interface requests the data for the frame (I-PDU) from the PDU Router at the point in time when it requires the data (i.e. right before sending the LIN frame).
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 onto such systems. Therefore it is not necessary to have an AUTOSAR system 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 refer 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.

Note: LIN slaves cannot fulfill the requirements to a Debugging Host, since LIN is not a multi-master bus.
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.

**Please Note:**
- There are two transport protocol modules in the FlexRay stack which can be used alternatively
  - FrTp: FlexRay ISO Transport Layer
  - FrArTp: FlexRay AUTOSAR Transport Layer, provides bus compatibility to AUTOSAR R3.x
Properties:

- Implementation: µC and ECU HW independent, partly dependent on FlexRay.
- AUTOSAR COM, Generic NM Interface and Diagnostic Communication Manager are the same for all vehicle network systems and exist as one instance per ECU.
- Generic NM Interface contains only a dispatcher. No further functionality is included. In case of gateway ECUs, it is replaced by the NM Coordinator which in addition provides the functionality to synchronize multiple different networks (of the same or different types) to synchronously wake them up or shut them down.
- FlexRay NM is specific for FlexRay networks and is instantiated per FlexRay vehicle network system.
- The communication system specific FlexRay State Manager handles the communication system dependent Start-up and Shutdown features. Furthermore it controls the different options of COM to send PDUs and to monitor signal timeouts.
The **TCP/IP Communication Services** are a group of modules for vehicle network communication with the communication system TCP/IP.

**Task:**
- Provide a uniform interface to the TCP/IP network. Hide protocol and message properties from the application.
Properties:

- The SoAd Module fully contains the DoIP protocol handler.
- The SoAd provides two APIs towards the TCP/IP stack.
- The TCP/IP Stack may contain a multitude of protocol handlers including, but not limited to UDP, TCP, DHCP, ARP, ICMP, and others.
- The TCP/IP stack is not intended as an AUTOSAR Module, but is defined by the interfaces only.
General communication stack properties:

- 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 on the bus).
- Upper Interface: μC, ECU hardware and network type independent.
- For refinement of GW architecture please refer to “Example Communication”
The **Memory Services** consist of one module, the NVRAM Manager. It is responsible for the management of non volatile data (read/write from different memory drivers).

**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)
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 (which includes timer services) and Error Manager.

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
There are dedicated modules for different aspects of error handling in AUTOSAR. E.g.:

- The **Debugging** module supports debugging of the AUTOSAR BSW. It interfaces to ECU internal modules and to an external host system via communication.
- The **Diagnostic Event Manager** is responsible for processing and storing diagnostic events (errors) and associated FreezeFrame data.
- The module **Diagnostic Log and Trace** supports logging and tracing of applications. It collects user defined log messages and converts them into a standardized format.
  - All detected development errors in the Basic Software are reported to **Development Error Tracer**.
  - The **Diagnostic Communication Manager** provides a common API for diagnostic services
  - etc.
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, relocatability is restricted.

**Task:**
Provide an abstraction from the specific physical properties of hardware sensors and actuators, which are connected to an ECU.

**Properties:**
- Implementation: µC and ECU HW independent, sensor and actuator dependent
Table of contents

1. Architecture
   1. Overview of Software Layers
   2. Content of Software Layers
   3. **Content of Software Layers in Multi Core systems**
   4. Overview of Modules
   5. Interfaces
      1. General
      2. Interaction of Layers (Examples)

2. Configuration

3. Integration and Runtime aspects
Example of a Layered Software Architecture for Multi-Core Microcontroller

ECU

core 0:

Complex Drivers

Microcontroller (µC)

core 1:

Application Layer

RTE

System Services

Memory Services

Communication Services

I/O HW Abstraction

Onboard Dev. Abstraction

Memory HW Abstraction

COM HW Abstraction

Complex Drivers

Operating System

ECU State Manager

Core Test

Example: an ECU with a two core microcontroller
The IOC, as shown in the figure, provides communication services which can be accessed by clients which need to communicate across OS-Application boundaries on the same ECU. The IOC is part of the OS.

Every Core runs a kind of ECU state management.
### Table of contents

1. **Architecture**
   1. Overview of Software Layers
   2. Content of Software Layers
   3. Content of Software Layers in Multi Core systems
   4. **Overview of Modules**
   5. Interfaces
      1. General
      2. Interaction of Layers (Examples)

2. Configuration

3. Integration and Runtime aspects
This figure shows the mapping of basic software modules to AUTOSAR layers.
The clustering shown in this document is the one defined by the project so far. AUTOSAR is currently not restricting the clustering on ICC2 level to dedicated clusters as many different constraint and optimization criteria might lead to different ICC2 clusterings. There might be different AUTOSAR ICC2 clusterings against which compliancy can be stated based on a to be defined approach for ICC2 compliance.
In a basic software which is compliant to ICC1 no modules or clusters are required.
The inner structure of this proprietary basic software is not specified.
Basic software (including the RTE) which is AUTOSAR compliant (ICC1-3) has to behave to the outside as specified by ICC3. For example the behavior towards:
- buses,
- boot loaders and
- applications.
# Table of contents

1. **Architecture**
   1. Overview of Software Layers
   2. Content of Software Layers
   3. Content of Software Layers in Multi Core systems
   4. Overview of Modules
   5. **Interfaces**
      1. General
      2. Interaction of Layers (Examples)

2. Configuration

3. Integration and Runtime aspects
### Interfaces
#### Type of Interfaces in AUTOSAR

<table>
<thead>
<tr>
<th>AUTOSAR Interface</th>
<th>An &quot;AUTOSAR Interface&quot; defines the information exchanged between software components and/or BSW modules. This description is independent of a specific programming language, ECU or network technology. AUTOSAR Interfaces are used in defining the ports of software-components and/or BSW modules. Through these ports software-components and/or BSW modules can communicate with each other (send or receive information or invoke services). AUTOSAR makes it possible to implement this communication between Software-Components and/or BSW modules either locally or via a network.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardized AUTOSAR Interface</td>
<td>A &quot;Standardized AUTOSAR Interface&quot; is an &quot;AUTOSAR Interface&quot; whose syntax and semantics are standardized in AUTOSAR. The &quot;Standardized AUTOSAR Interfaces&quot; are typically used to define AUTOSAR Services, which are standardized services provided by the AUTOSAR Basic Software to the application Software-Components.</td>
</tr>
<tr>
<td>Standardized Interface</td>
<td>A &quot;Standardized Interface&quot; is an API which is standardized within AUTOSAR without using the &quot;AUTOSAR Interface&quot; technique. These &quot;Standardized Interfaces&quot; are typically defined for a specific programming language (like &quot;C&quot;). Because of this, &quot;standardized interfaces&quot; are typically used between software-modules which are always on the same ECU. When software modules communicate through a &quot;standardized interface&quot;, it is NOT possible any more to route the communication between the software-modules through a network.</td>
</tr>
</tbody>
</table>
Interfaces
Components and interfaces view (simplified)

Note: This figure is incomplete with respect to the possible interactions between the layers.

- AUTOSAR Confidential -
**Interfaces: General Rules**

**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

- **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.**
**Interfaces: General Rules**

**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>

- AUTOSAR Confidential -
Complex Drivers may need to interface to other modules in the layered software architecture, or modules in the layered software architecture may need to interface to a Complex Driver. If this is the case, the following rules apply:

1. **Interfacing from modules of the layered software architecture to Complex Drivers**

This is only allowed if the Complex Driver offers an interface which can be generically configured by the accessing AUTOSAR module.

A typical example is the PDU Router: a Complex Driver may implement the interface module of a new bus system. This is already taken care of within the configuration of the PDU Router.

2. **Interfacing from a Complex Driver to modules of the layered software architecture**

Again, this is only allowed if the respective modules of the layered software architecture offer the interfaces, and are prepared to be accessed by a Complex Driver. Usually this means that

- The respective interfaces are defined to be re-entrant.
- If call back routines are used, the names are configurable
- No upper module exists which does a management of states of the module (parallel access would change states without being noticed by the upper module)
**Interfaces**

**Interfacing with Complex Drivers (2)**

In general, it is possible to access the following modules:

- The SPI driver
- The GPT driver
- The I/O drivers with the restriction that re-entrancy often only exists for separate groups/channels/etc. Parallel access to the same group/channel/etc. is mostly not allowed. This has to be taken care of during configuration.
- The NVRAM Manager as exclusive access point to the memory stack
- The Watchdog Manager as exclusive access point to the watchdog stack
- The PDU Router as exclusive bus and protocol independent access point to the communication stack
- The bus specific interface modules as exclusive bus specific access point to the communication stack
- The NM Interface Module as exclusive access point to the network management stack
- The Communication Manager (only from upper layer) and the Basic Software Mode Manager as exclusive access points to state management
- Det, Dem and Dlt
- The OS as long as the used OS objects are not used by a module of the layered software architecture

Still, for each module it is necessary to check if the respective function is marked as being re-entrant. For example, ‘init’ functions are usually not re-entrant and should only be called by the ECU State Manager.
**Interfaces**

**Interfacing with Complex Drivers (3)**

In case of multi core architectures, there are additional rules:

- In AUTOSAR release 4.0, the BSW resides on the master core.
- Consequently, if the CDD needs to access standardized interfaces of the BSW, it needs to reside on the same core.
- In case a CDD resides on a different core, it can use the normal port mechanism to access AUTOSAR interfaces and standardized AUTOSAR interfaces. This invokes the RTE, which uses the IOC mechanism of the operating system to transfer requests to the master core.
- However, if the CDD needs to access standardized interfaces of the BSW and does not reside on the master core, a stub part of the CDD needs to be implemented on the master core, and communication needs to be organized CDD-local using the IOC mechanism of the operating system similar to what the RTE does.
- Additionally, in this case the initialization part of the CDD also needs to reside in the stub part on the master core.

---

![Complex Drivers Diagram](image)

**Microcontroller (μC)**

<table>
<thead>
<tr>
<th>Application Layer</th>
<th>RTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Services</td>
<td>Memory Services</td>
</tr>
<tr>
<td>Memory Drivers</td>
<td>Communication Services</td>
</tr>
<tr>
<td>Communication Drivers</td>
<td>I/O Drivers</td>
</tr>
</tbody>
</table>

**Complex Drivers**

- Onboard Dev. Abstr.
- Memory HW Abstr.
- COM HW Abstr.
- I/O Drivers

**Microcontroller (µC)**

- Communication Drivers
- Memory Abstraction
- Memory Services
## Table of contents

1. **Architecture**
   1. Overview of Software Layers
   2. Content of Software Layers
   3. Content of Software Layers in Multi Core systems
   4. Overview of Modules
   5. **Interfaces**
      1. General
      2. *Interaction of Layers (Examples)*

2. Configuration
3. Integration and Runtime aspects
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:

The ECU hardware includes an external EEPROM 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 includes 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“)
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,
    uint16  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_ReturnType Ea_Write
(
    uint16  BlockNumber,
    uint8   *DataBufferPtr
)
```
**Interfaces: Interaction of Layers – Example “Memory”**

Implementation of Memory Abstraction Interface

**Situation 1: only one NV device type used**

This is the usual use case. In this situation, the Memory Abstraction can, in case of source code availability, be implemented as a simple macro which neglects the `DeviceIndex` parameter. The following example shows the write function only:

**File MemIf.h:**
```
#include "Ea.h" /* for providing access to the EEPROM Abstraction */
...
#define MemIf_Write(DeviceIndex, BlockNumber, DataBufferPtr) \n  Ea_Write(BlockNumber, 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.
Interfaces: Interaction of Layers – Example “Memory”
Implementation of Memory Abstraction Interface

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 an array of pointers to function. The following example shows the write function only:

File MemIf.h:

```c
extern const WriteFctPtrType WriteFctPtr[2];

#define MemIf_Write(DeviceIndex, BlockNumber, DataBufferPtr) 
WriteFctPtr[DeviceIndex](BlockNumber, 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.
Conclusions:

- Abstraction Layers can be implemented very efficiently

- Abstraction Layers can be scaled

- The Memory Abstraction Interface eases the access of the NVRAM Manager to one or more EEPROM and Flash devices
Interfaces: Interaction of Layers – Example “Communication”

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.
**Interfaces: Interaction of Layers**

**Example “Communication” (1)**

**SDU and PDU Naming Conventions**

The 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 / TTCAN prefix</th>
<th>LIN prefix</th>
<th>FlexRay prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 6:</td>
<td>I</td>
<td>COM, DCM</td>
<td>I-PDU</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation (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>N</td>
<td>TP Layer</td>
<td>N-PDU</td>
<td>CAN SF</td>
<td>LIN SF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer 3:</td>
<td></td>
<td></td>
<td>CAN FF</td>
<td>LIN FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Layer</td>
<td></td>
<td></td>
<td>CAN CF</td>
<td>LIN CF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer 2:</td>
<td>L</td>
<td>Driver, Interface</td>
<td>L-PDU</td>
<td>CAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Link Layer</td>
<td></td>
<td></td>
<td>CAN FC</td>
<td>LIN FC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CAN</td>
<td>LIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FR</td>
<td>FR</td>
<td></td>
<td></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

For details on the frame types, please refer to the AUTOSAR Transport Protocol specifications for CAN, TTCAN, LIN and FlexRay.
Interfaces: Interaction of Layers
Example “Communication” (2)

Components

➢ PDU Router:
  ■ 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 Coordinator:
  ■ Synchronization of Network States of different communication channels connected to an ECU via the network managements handled by the NM Coordinator

➢ Communication State Managers:
  ■ Start and Shutdown the hardware units of the communication systems via the interfaces
  ■ Control PDU groups
Interfaces: Interaction of Layers
Example “Communication” (3)

1 The Interface between PduR and Tp differs significantly compared to the interface between PduR and the Ifs. In case of TP involvement a handshake mechanism is implemented allowing the transmission of I-Pdus > Frame size.
2 CanIf with TTCAN serves both CanDrv with or without TTCAN. CanIf without TTCAN cannot serve CanDrv with TTCAN.
This figure shows the interaction of and inside the internet protocol stack.
Table of contents

1. Architecture
   1. Overview of Software Layers
   2. Content of Software Layers
   3. Content of Software Layers in Multi Core systems
   4. Overview of Modules
   5. Interfaces
      1. General
      2. Interaction of Layers (Examples)

2. Configuration
3. Integration and Runtime aspects
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)
   - Single or multiple configuration sets can be provided. In case that multiple configuration sets are provided, the actually used configuration set is to be specified at runtime.

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.
Configuration
Pre-compile time (1)

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 two
configuration files (`*_Cfg.h, *_Cfg.c`) and/or by code generation:

- `*_Cfg.h` stores e.g. macros and/or `#defines`
- `*_Cfg.c` stores e.g. constants
Example 1: Enabling/disabling functionality

File Spi_Cfg.h:
#define SPI_DEV_ERROR_DETECT   ON

File Spi_Cfg.c:
const uint8 myconstant = 1U;

File Spi.c (available as source code):
#include "Spi_Cfg.h"     /* for importing the configuration parameters */
extern const uint8 myconstant;

#if (SPI_DEV_ERROR_DETECT == ON)
Det_ReportError(Spi_ModuleId, 0U, 3U, SPI_E_PARAM_LENGTH);  /* only one instance available */
#endif

Note: The Compiler Abstraction and Memory Abstraction (as specified by AUTOSAR) are not used to keep the example simple.
Example 2: Event IDs reported to the Dem
XML configuration file of the NVRAM Manager:
Specifies that it needs the event symbol `NVM_E_REQ_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 DemConf_DemEventParameter_FLS_E_ERASE_FAILED_0   1U
#define DemConf_DemEventParameter_FLS_E_ERASE_FAILED_1   2U
#define DemConf_DemEventParameter_FLS_E_WRITE_FAILED_0   3U
#define DemConf_DemEventParameter_FLS_E_WRITE_FAILED_1   4U
#define DemConf_DemEventParameter_NVM_E_REQ_FAILED       5U
#define DemConf_DemEventParameter_CANSM_E_BUS_OFF         6U
...
```

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_ReportErrorStatus(DemConf_DemEventParameter_NVM_E_REQ_FAILED, DEM_EVENT_STATUS_PASSED);
```

Example for a multiple instance driver (e.g. internal and external flash module)
Use cases
Link time configuration would be chosen for
- Configuration of modules that are only available as object code (e.g. IP protection or warranty reasons)
- Selection of configuration set after compilation but before linking.

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 multiple instantiated module (Flash Driver) only available as object code XML configuration file of the Flash Driver:
Specifies that it needs the event symbol **FLS_E_WRITE_FAILED** for production error reporting.

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

```c
#define DemConf_DemEventParameter_FLS_E_WRITE_FAILED_0   1U
#define DemConf_DemEventParameter_FLS_E_WRITE_FAILED_1   2U
#define DemConf_DemEventParameter_FLS_E_WRITE_FAILED_0   3U
#define DemConf_DemEventParameter_FLS_E_WRITE_FAILED_1   4U
#define DemConf_DemEventParameter_NVME_WRITE_FAILED      5U
#define DemConf_DemEventParameter_CANSM_E_BUS_OFF         6U
...`

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

const Dem_EventIdType Fls_WriteFailed[2] = {
  DemConf_DemEventParameter_FLS_E_WRITE_FAILED_1,
  DemConf_DemEventParameter_FLS_E_WRITE_FAILED_2};
```

File Fls.c (available as object code):
```c
#include "Dem.h" /* for reporting production errors */
extern const Dem_EventIdType Fls_WriteFailed[];

Dem_ReportErrorStatus(Fls_WriteFailed[instance], DEM_EVENT_STATUS_FAILED);
```

**Note**: the complete include file structure with all forward declarations is not shown here to keep the example simple.
Example 2: Event IDs reported to the Dem by a module (Flash Driver) 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 Flash Driver 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.
**Configuration**

**Post-build time (1)**

**Use cases**
Post-build time configuration would be chosen for
- Configuration of data where only the structure is defined but the contents not known during ECU-build time
- 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 has impact on performance, code and data 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.

2. 1..n configuration sets, 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 (Post-build time loadable)**

If the configuration data is fix in memory size and position, the module has direct access to these external structs.

```
<table>
<thead>
<tr>
<th>PduR.c</th>
<th>Compiler</th>
<th>Linker</th>
<th>Linker control file</th>
</tr>
</thead>
<tbody>
<tr>
<td>PduR_PBcfg.c</td>
<td>Compiler</td>
<td>Linker</td>
<td>PduR.o</td>
</tr>
</tbody>
</table>
```

Direct access (via reference as given by the pointer parameter of PduR's initialization function)
Required implementation 2: Configuration of CAN Driver that is available as object code only; multiple configuration sets can be selected during initialization time.

File Can_PBcfg.c:
#include "Can.h" /* for getting Can_ConfigType */
const Can_ConfigType MySimpleCanConfig [2] =
{
    
    Can_BitTiming = 0xDF,
    Can_AcceptanceMask1 = 0xFFFFFFFF,
    Can_AcceptanceMask2 = 0xFFFFFFFF,
    Can_AcceptanceMask3 = 0x00034DFF,
    Can_AcceptanceMask4 = 0x00FF0000
},
    
    {    ...    }
};

File EcuM.c:
#include "Can.h" /* for initializing the CAN Driver */
Can_Init(&MySimpleCanConfig[0]);

File Can.c (available as object code):
#include "Can.h" /* for getting Can_ConfigType */

void Can_Init(Can_ConfigType* Config)
{
    /* write the init data to the CAN HW */
};
Different use cases require different kinds of configurability. Therefore the following configuration variants are provided:

- **VARIANT-PRE-COMPILE**
  Only parameters with "Pre-compile time" configuration are allowed in this variant.

- **VARIANT-LINK-TIME**
  Only parameters with "Pre-compile time" and "Link time" are allowed in this variant.

- **VARIANT-POST-BUILD**
  Parameters with "Pre-compile time", "Link time" and "Post-build time" are allowed in this variant.

**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, 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 class 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).

- AUTOSAR Confidential -
Configuration
Memory Layout Example: Postbuild Loadable (PBL)

EcuM defines the index:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>&amp;index (=0x8000)</td>
</tr>
<tr>
<td>0x8000</td>
<td>&amp;xx_configuration = 0x4710</td>
</tr>
<tr>
<td>0x8002</td>
<td>&amp;yy_configuration = 0x4720</td>
</tr>
<tr>
<td>0x8004</td>
<td>&amp;zz_configuration = 0x4730</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Xx defines the modules configuration data:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4710</td>
<td>&amp;the_real_xx_configuration</td>
</tr>
<tr>
<td>0x4710</td>
<td>lower = 2</td>
</tr>
<tr>
<td>0x4712</td>
<td>upper = 7</td>
</tr>
<tr>
<td>0x4714</td>
<td>more_data</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Yy defines the modules configuration data:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4720</td>
<td>&amp;the_real_yy_configuration</td>
</tr>
<tr>
<td>0x4720</td>
<td>Xx_data1=0815</td>
</tr>
<tr>
<td>0x4722</td>
<td>Yy_data2=4711</td>
</tr>
<tr>
<td>0x4724</td>
<td>more_data</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Description where to find what is an overall agreement:
1. EcuM needs to know all addresses including index
2. The modules (xx, yy, zz) need to know their own start address: in this case: 0x4710, 0x4720 ...
3. The start addresses might be dynamic i.e. changes with new configuration
4. When initializing a module (e.g. xx, yy, zz), EcuM passes the base address of the configuration data (e.g. 0x4710, 0x4720, 0x4730) to the module to allow for variable sizes of the configuration data.

The module data is agreed locally (in the module) only
1. The module (xx, yy) knows its own start address (to enable the implementer to allocate data section)
2. Only the module (xx, yy) knows the internals of its own configuration

For details, see Chapter “Post build implementation” in “AUTOSAR_TR_CLImplementationRules.pdf”
### Configuration

#### Memory Layout Example: Postbuild Multiple Selectable (PBM)

<table>
<thead>
<tr>
<th>FL</th>
<th>xx_configuration = 0x4710</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>&amp;index[] (=0x8000)</td>
</tr>
<tr>
<td>0x8002</td>
<td>&amp;yy_configuration = 0x4720</td>
</tr>
<tr>
<td>0x8004</td>
<td>&amp;zz_configuration = 0x4730</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0x8008</td>
<td>&amp;xx_configuration = 0x5000</td>
</tr>
<tr>
<td>0x800a</td>
<td>&amp;yy_configuration = 0x5400</td>
</tr>
<tr>
<td>0x800c</td>
<td>&amp;zz_configuration = 0x5200</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0x8010</td>
<td>&amp;xx_configuration = ...</td>
</tr>
<tr>
<td>0x8012</td>
<td>&amp;yy_configuration = ...</td>
</tr>
<tr>
<td>0x8014</td>
<td>&amp;zz_configuration = ...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

As before, the description where to find what is an overall agreement

1. The index contains more than one description (FL, FR, ...) in an array (here the size of an array element is agreed to be 8)

2. There is an agreed variable containing the position of one description selector = CheckPinCombination()

3. Instead of passing the pointer directly there is one indirection: (struct EcuM_ConfigType *) &index[selector];

4. Everything else works as in PBL

For details, see Chapter “Post build implementation” in “AUTOSAR_TR_CImplementationRules.pdf”
# Table of contents

1. Architecture
2. Configuration

3. Integration and Runtime aspects
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. Debugging
   7. Measurement and Calibration
   8. Functional Safety
   9. Energy Management
Integration and Runtime aspects
Mapping of Runnables

- Runnables are the active parts of Software Components
- They can be executed concurrently, by mapping them to different Tasks.
- The figure shows further entities like OS-applications, Partitions, µC-Cores and BSW-Resources which have to be considered for this mapping.
Table of contents

1. Architecture
2. Configuration
3. Integration and Runtime aspects
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. Debugging
   7. Measurement and Calibration
   8. Functional Safety
   9. Energy Management
Integration and Runtime aspects - Partitioning

Introduction

- Partitions are used as error containment regions:
  - Permit logical grouping of SW-Cs and resources
  - Recovery policies defined individually for each partition

- Partition consistency is ensured by the system/platform, for instance for:
  - Memory access violation
  - Time budget violation

- Partitions can be terminated or restarted during run-time as a result of a detected error:
  - Further actions required: see example on following slides
  - The BSW is placed in a privileged partition
  - This partition should not be restarted or terminated

- Partitions are configured in the ECU-C:
  - A partition is implemented by an OS-Application within the OS
  - SW-Cs are mapped to partitions (Consequence: restricts runnable to task mapping)
  - A partition can be configured as restartable or not

- Communication across partitions is realized by the IOC
Integration and Runtime aspects - Partitioning
Example of restarting partition

A violation (error) has occurred in the system (e.g., memory or timing violation)

Decision (by integrator code) to restart the partition

Other partitions remain unaffected

The partition is terminated by the OS, cleanup possible

Communication to the partition is stopped

Communication from the partition is stopped (e.g., default values for ports used)

The partition is restarting (integrator code), initial environment for partition setup (init runnables, port values etc)

Communication to the partition is stopped

Communication from the partition is stopped

The partition is restarted and up and running

Communication is restored

Partition internally handles state consistency
Integration and Runtime aspects - Partitioning

Involved components

- Protection Hook
  - Executed on protection violation (memory or timing)
  - Decides what the action is (Terminate, Restart, Shutdown, Nothing)
  - Provided by integrator
  - OS acts on decision by inspecting return value

- OsRestartTask
  - Started by OS in case Protection Hook returns Restart
  - Provided by integrator
  - Runs in the partition’s context and initiates necessary cleanup and restart activities, such as:
    - Stopping communication (ComM)
    - Updating NvM
    - Informing Watchdog, CDDs etc.

- RTE
  - Functions for performing cleanup and restart of RTE in partition
  - Triggers init runnables for restarted partition
  - Handles communication consistency for restarting/terminated partitions

- Operating System
  - OS acts on OS-Applications, which are containers for partitions
  - OS-Applications have states (APPLICATION_ACCESSIBLE, APPLICATION_RESTART, APPLICATION_TERMINATED)
  - OS provides API to terminate other OS-Applications (for other errors than memory/timing)
Integration and Runtime aspects - Partitioning
restart example

Os-Application state for the considered Partition.

APPLICATION_ACTIVE

APPLICATION_RESTARTING

APPLICATION_ACTIVE

- AUTOSAR Confidential -
**Integration and Runtime aspects - Partitioning**

**Other examples**

- **Termination**
  - A partition can be terminated directly
  - Also for termination, some cleanup may be needed, and this shall be performed in the same way as when restarting a partition
  - No restart is possible without a complete ECU restart

- **Error detection in applications**
  - Application-level SW-Cs may detect errors (i.e., not memory/timing)
  - A termination/restart can be triggered from a SW-C using the OS service `TerminateApplication()`
  - Example: a distributed application requires restart on multiple ECUs
<table>
<thead>
<tr>
<th>Table of contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Architecture</td>
</tr>
<tr>
<td>2. Configuration</td>
</tr>
<tr>
<td>3. <strong>Integration and Runtime aspects</strong></td>
</tr>
<tr>
<td>1. Mapping of Runnables</td>
</tr>
<tr>
<td>2. Partitioning</td>
</tr>
<tr>
<td>3. <strong>Scheduling</strong></td>
</tr>
<tr>
<td>4. Mode Management</td>
</tr>
<tr>
<td>5. Error Handling, Reporting and Diagnostic</td>
</tr>
<tr>
<td>6. Debugging</td>
</tr>
<tr>
<td>7. Measurement and Calibration</td>
</tr>
<tr>
<td>8. Functional Safety</td>
</tr>
<tr>
<td>9. Energy Management</td>
</tr>
</tbody>
</table>
Integration and Runtime aspects - Scheduling

General Architectural Aspects

- Basic Software Scheduler and the RTE are generated together.
- This enables
  - that the same OS Task schedules BSW Main Functions and Runnable Entities of Software Components
    - to optimize the resource consumption
    - to configure interlaced execution sequences of Runnable Entities and BSW Main functions.
  - a coordinated switching of a Mode affecting BSW Modules and Application Software Components
  - the synchronized triggering of both, Runnable Entities and BSW Main Functions by the same External Trigger Occurred Event.
Integration and Runtime aspects - Scheduling
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 mechanisms
- Applied data consistency mechanisms shall be configured by the ECU/BSW integrator dependent from the configured scheduling.

Single BSW modules do not know about
- ECU wide timing dependencies
- Scheduling implications
- Most efficient way to implement data consistency

Centralize the BSW schedule in the BSW Scheduler configured by the ECU/BSW integrator and generated by the RTE generator together with the RTE
- Eases the integration task
- Enables applying different scheduling strategies to schedulable objects
  - Preemptive, non-preemptive, ...
- Enables applying different data consistency mechanisms
- Enables reducing resources (e.g., minimize the number of tasks)
- Enables interlaced execution sequences of Runnable Entities and BSW Main functions

Restrict the usage of OS functionality
- Only the BSW Scheduler and the RTE shall use OS objects or OS services (exceptions: EcuM, Complex Drivers and services: GetCounterValue and GetElapsedTimeCounterValue of OS; MCAL modules may enable/disable interrupts )
- Rationale:
  - Scheduling of the BSW shall be transparent to the system (integrator)
  - Enables reducing the usage of OS resources (Tasks, Resources,...)
  - Enables re-using modules in different environments
Integration and Runtime aspects - Scheduling
Scheduling Objects, Triggers and Mode Disabling Dependencies

BSW Scheduling objects
- Main functions
  - n per module
  - located in all layers

BSW Events
- BswTimingEvent
- BswBackgroundEvent
- BswModeSwitchEvent
- BswModeSwitchedAckEvent
- BswInternalTriggerOccuredEvent
- BswExternalTriggerOccuredEvent

Triggers
- Main functions can be triggered in all layers by the listed BSW Events

Mode Disabling Dependencies
- The scheduling of Main functions can be disabled in particular modes.
# Integration and Runtime aspects - Scheduling Transformation Process

## Logical Architecture (Model)
- Ideal concurrency
- Unrestricted resources
- Only real data dependencies
- Scheduling objects
- Trigger
  - BSW Events
- Sequences of scheduling objects
- Scheduling Conditions
- ...

## Technical Architecture (Implementation)
- Restricted concurrency
- Restricted resources
- Real data dependencies
- Dependencies given by restrictions
- 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 -
Integration and Runtime aspects - Scheduling
Transformation Process – Example 1

Logical Architecture (Model)

- Zzz_MainFunction_Bbb();
- Yyy_MainFunction_Aaa();
- Xxx_MainFunction_Aaa();

Technical Architecture (Schedule Module)

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

- Mapping of scheduling objects to OS Tasks
- Specification of sequences of scheduling objects within tasks

- AUTOSAR Confidential -
Integration and Runtime aspects - Scheduling
Transformation Process – Example 2

Logical Architecture (Model)

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

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

Technical Architecture (Schedule Module)

- Mapping of scheduling objects to OS Tasks
Integration and Runtime aspects - Scheduling
Data Consistency – Motivation

Logical Architecture (Model)  

- 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();

- Technical Architecture (Schedule Module)

- Technical Architecture (Schedule Module)
  - Yyy_Module
    - Yyy_MainFunction();

- Yyy_AccessResource();
- XYZ resource

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

Transformation
Integration and Runtime aspects - Scheduling
Data Consistency – Example 1 – “Critical Sections” Approach

Logical Architecture (Model)/
Technical Architecture (Schedule Module)

Task1    Xxxx_Module
        
        Xxxx_MainFunction();

Yyy_AccessResource();

XYZ resource

Yyy_MainFunction();

Task2

Implementation of Schedule Module

#define SchM_Enter_<mod>_<name>  
    DisableAllInterrupts
#define SchM_Exit_<mod>_<name>   
    EnableAllInterrupts

Yyy_AccessResource() {
    ...
    SchM_Enter_Xxx_XYZ();
    <access_to_shared_resource>
    SchM_Exit_Xxx_XYZ();
    ...
}

Yyy_MainFunction() {
    ...
    SchM_Enter_Yyy_XYZ();
    <access_to_shared_resource>
    SchM_Exit_Yyy_XYZ();
    ...
}

Data consistency is ensured by:
- Interrupt blocking
Integration and Runtime aspects - Scheduling

Data Consistency – Example 1 – "Critical Sections" Approach

Logical Architecture (Model)/
Technical Architecture (Schedule Module)

Task1  Xxx_Module

Xxx_MainFunction();

Yyy_AccessResource();

XYZ resource

Yyy_MainFunction();

Task2

Implementation of Schedule Module

```c
#define SchM_Enter_<mod>_<name> /* nothing required */
#define SchM_Exit_<mod>_<name> /* nothing required */

Yyy_AccessResource() {
    ...
    SchM_Enter_Xxx_XYZ();
    <access_to_shared_resource>
    SchM.Exit_Xxx_XYZ();
    ...
}

Yyy_MainFunction() {
    ...
    SchM.Enter_Yyy_XYZ();
    <access_to_shared_resource>
    SchM.Exit_Yyy_XYZ();
    ...
}
```

Data consistency is ensured by:
- Sequence
Integration and Runtime aspects
Mode Communication / Mode Dependent Scheduling

- The mode dependent scheduling of BSW Modules is identical to the mode dependent scheduling of runnables of software components.
- A mode manager defines a Provide \texttt{ModeDeclarationGroupPrototype} in its Basic Software Module Description, and the BSW Scheduler provides an API to communicate mode switch requests to the BSW Scheduler.
- A mode user defines a Required \texttt{ModeDeclarationGroupPrototype} in its Basic Software Module Description. On demand the BSW Scheduler provides an API to read the current active mode.
- If the Basic Software Module Description defines Mode Disabling Dependencies, the BSW Scheduler suppresses the scheduling of BSW Main functions in particular modes.
Table of contents

1. Architecture
2. Configuration
3. Integration and Runtime aspects
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. Debugging
   7. Measurement and Calibration
   8. Functional Safety
   9. Energy Management
**Integration and Runtime aspects**

**Vehicle and application mode management (1)**

**Relation of Modes:**

- Every system contains Modes at different levels of granularity. As shown in the figure, there are vehicle modes and several applications with modes and ECUs with local BSW modes.
- Modes at all these levels influence each other.

Therefore:

- Depending on vehicle modes, applications may be active or inactive and thus be in different application modes.
- Vice versa, the operational state of certain applications may cause vehicle mode changes.
- Depending on vehicle and application modes, the BSW modes may change, e.g., the communication needs of an application may cause a change in the BSW mode of a communication network.
- Vice versa, BSW modes may influence the modes of applications and even the whole vehicle, e.g., when a communication network is unavailable, applications that depend on it may change into a limp-home mode.
Processing of Mode Requests

The basic idea of vehicle mode management is to distribute and arbitrate mode requests and to control the BSW locally based on the results.

This implies that on each ECU, there has to be a mode manager that switches the modes for its local mode users and controls the BSW. Of course there can also be multiple mode managers that switch different Modes.

The mode request is a “normal” sender/receiver communication (system wide) while the mode switch always a local service.
Integration and Runtime aspects
Vehicle and application mode management (3)

- The major part of the needed functionality is placed in a new BSW module which we call the Basic Software Mode Manager (BswM for short). Since the BswM is located in the BSW, it is present on every ECU and local to the mode users as well as the controlled BSW modules.

- The distribution of mode requests is performed by the RTE and the RTE also implements the handling of mode switches.

- E.g. for vehicle modes, a mode request originates from one central mode requestor SW-C and has to be received by the BswMs in many ECUs. This is an exception of the rule that SW-Cs may only communicate to local BSW modules.
Integration and Runtime aspects
Vehicle and application mode management (4)

Mode Processing Cycle

- The mode requester SW-C requests mode A through its sender port. The RTE distributes the request and the BswM receives it through its receiver port.
- The BswM evaluates its rules and if a rule triggers, it executes the corresponding action list.
- When executing the action list, the BswM may issue a (configurable optional) RTE call to the mode switch API as a last action to inform the mode users about the arbitration result, e.g. the resulting mode A’.
- Any SW-C, especially the mode requester can register to receive the mode switch indication.
- The mode requests can originate from local and remote ECUs.
- Note that the mode requestor can only receive the mode switch indications from the local BswM, even if the requests are sent out to multiple ECUs.

- AUTOSAR Confidential -
**Table of contents**

1. Architecture
2. Configuration

3. **Integration and Runtime aspects**
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. **Error Handling, Reporting and Diagnostic**
   6. Debugging
   7. Measurement and Calibration
   8. Functional Safety
   9. Energy Management
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

System errors
- Example 1: CAN receive buffer overflow
- Example 2: time-out for receive messages
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic
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
- Functional degradation
- Loss of functionality

Failure mode in terms of time
- Permanent errors
- Transient / sporadic errors
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic

Error Reporting – Alternatives

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.

There are several alternatives to report an error (detailed on the following slides):

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

Via statically definable callback function (notification)
Inform the caller about failure of an operation

Via central Error Hook (Development Error Tracer)
For logging and tracing errors during product development. Can be switched off for production code.

Via central Error Function (AUTOSAR Diagnostic Event Manager)
For error reaction and logging in series (production code)

Each application software component (SW-C) can report errors to the Dlt.
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic Mechanism in relation to AUTOSAR layers and system life time

Life cycle:

- development
- production
- After production
Error reporting via API
Informs the caller about failure of an operation by returning an error status.

Basic return type
Success: $E_{\text{OK}}$ (value: 0)
Failure: $E_{\text{NOT_OK}}$ (value: 1)

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

Example: services of EEPROM driver
Success: $EEP_{\text{E_OK}}$
General failure (service not accepted): $EEP_{\text{E_NOT_OK}}$
Write Operation to EEPROM was not successful: $EEP_{\text{E_WRITE_FAILED}}$
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic

Error Reporting – Introduction

Error reporting via Diagnostic Event Manager (Dem)
For reporting production / series errors.
Those errors have a defined reaction depending on the configuration of this ECU, e.g.:
- Writing to error memory
- Disabling of ECU functions (e.g. via Function Inhibition Manager)
- Notification of SW-Cs

The Diagnostic Event Manager is a standard AUTOSAR module which is always available in production code and whose functionality is specified within AUTOSAR.

Error reporting via Development Error Tracer (Det)
For reporting development errors.
The Development Error Tracer is mainly intended for tracing and logging errors during development. Within the Development 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 specified within AUTOSAR, but the functionality can be chosen/implemented by the developer according to his specific needs.

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).
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic
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:
- Each BSW Module declares all production code relevant error variables it needs as “extern”
- Each BSW Module stores all error variables that it needs in the ECU configuration description (e.g. CANSM_E_BUS_OFF)
- 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 DemConf_DemEventParameter_CANSM_E_BUS_OFF=7U; or
  #define DemConf_DemEventParameter_CANSM_E_BUS_OFF ((Dem_EventIdType)7))
- The reaction to the errors is also defined in the Error Manager configuration tool. This configuration is project specific.
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic
Error Reporting – Development Error Tracer

API
The Development Error Tracer has syntactically the following API:

\[
\text{Det\_ReportError} \left( \text{uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId} \right)
\]

Error numbering concept

ModuleId (uint16)
The Module ID 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.

InstanceId (uint8)
The Instance ID represents the identifier of an indexed based module starting from 0. If the module is a single instance module it shall pass 0 as an instance ID.

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 BSW description.
Integration and Runtime aspects - Error Handling, Reporting and Diagnostic

Error Reporting – Diagnostic Log and Trace (1)

The module **Diagnostic Log and Trace (Dlt)** collects log messages and converts them into a standardized format. The Dlt forwards the data to the Dcm or a CDD which uses a serial interface for example.

Therefore the Dlt provides the following functionalities:

- **Logging**
  - logging of errors, warnings and info messages from AUTOSAR SW-Cs, providing a standardized AUTOSAR interface,
  - gathering all log and trace messages from all AUTOSAR SW-Cs in a centralized AUTOSAR service component (Dlt) in the BSW,
  - logging of messages from Det and
  - logging of messages from Dem.

- **Tracing**
  - of RTE activities

- **Control**
  - individual log and trace messages can be enabled/disabled and
  - Log levels can be controlled individually by back channel.

- **Generic**
  - Dlt is available during development and production phase,
  - access over standard diagnosis or platform specific test interface is possible and
  - security mechanisms to prevent misuse in production phase are provided.
The Dlt communication module is enabled by an external client.

The **external client** has to set up a diagnostic session in a defined security level and sending control message to Dlt for enabling the Dlt communication module.

A **SW-C** is generating a log message. The log message is sent to Dlt by calling the API provided by Dlt.

Dlt sends the log message to the implemented Dlt communication module interface.

At the end, the log message is stored on an external client and can be analyzed later on.
API

The Diagnostic Log and Trace has syntactically the following API:

```c
Dlt_SendLogMessage(Dlt_SessionIDType session_id, Dlt_MessageLogInfoType log_info, uint8 *log_data, uint16 log_data_length)
```

Log message identification:

- **session_id**
  - Session ID is the identification number of a log or trace session. A session is the logical entity of the source of log or trace messages. If a SW-C is instantiated several times or opens several ports to Dlt, a new session with a new Session ID for every instance is used. A SW-C additionally can have several log or trace sessions if it has several ports opened to Dlt.

- **log_info** contains:
  - Application ID / Context ID
    - Application ID is a short name of the SW-C. It identifies the SW-C in the log and trace message. Context ID is a user defined ID to group log and trace messages produced by a SW-C to distinguish functionality. Each Application ID can own several Context IDs. Context ID’s are grouped by Application ID’s. Both are composed by four 8 bit ASCII characters.
  - Message ID
    - Message ID is the ID to characterize the information, which is transported by the message itself. It can be used for identifying the source (in source code) of a message and shall be used for characterizing the payload of a message. A message ID is statically fixed at development or configuration time.

- **log_data**
  - Contain the log or trace data itself. The content and the structure of this provided buffer is specified by the Dlt transmission protocol.

Description File

Normally the **log_data** contains only contents of not fixed variables or information (e.g. no static strings are transmitted). Additionally a description file shall be provided. Within this file the same information for a log messages associated with the Message ID are posted. These are information how to interpret the **log_data** buffer and what fixed entries belonging to a log message.

- AUTOSAR Confidential -
# Table of contents

1. Architecture
2. Configuration

3. **Integration and Runtime aspects**
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. **Debugging**
   7. Measurement and Calibration
   8. Functional Safety
   9. Energy Management
The goal of the **Debugging Module** is to support a user (system integrator or BSW developer) during development, in case the basic software does not behave as expected. To do so, it collects as much information as possible about the runtime behavior of the systems without halting the processor. This data is transmitted to an external host system via communication, to enable the user to identify the source of a problem. An internal buffer is provided to decouple data collection from data transmission.

Main tasks of the Debugging Module are to

- Collect and store data for tracing purposes
- Collect and immediately transmit data to host
- Modify data in target memory on host request
- Transmit stored data to host
- Accept commands to change the behavior of the Debugging Module
The Debugging Module can:
- interface to ECU internal modules and to an external host system via communication.

With respect to the host system:
- the Debugging Module is also described as being ‘target’.

Internally, the Debugging Module consists of:
- a core part, which handles data sampling, and
- a communication part, which is responsible for transmission and reception of data.

The Debugging Module is designed to be:
- hardware independent and
- interfaces to the PDU router.

It can be used by:
- the BSW and
- RTE.

There is no interface to software components.
Integration and Runtime aspects - Debugging
External architectural view - Data flow host → target

Example:

- Host
- Debugging Com Module
- Debugging Core Module
- BSW

host command → call debug core command → collection of data

processing of host commands → confirmation
Integration and Runtime aspects - Debugging
External architectural view - Data flow target → host

Example:

- Debugging Com Module contains implementation of simplified TP
- TX request
- Collection of data

message fragment 1
message fragment 2
message fragment n
### Table of Contents

1. Architecture

2. Configuration

3. **Integration and Runtime aspects**
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. Debugging
   7. **Measurement and Calibration**
   8. Functional Safety
   9. Energy Management
Integration and Runtime aspects - Measurement and Calibration

XCP

**XCP** is an ASAM standard for calibration purpose of an ECU.

**XCP** within AUTOSAR provides the following basic **features**:

- Synchronous data acquisition
- Synchronous data stimulation
- Online memory calibration (read / write access)
- Calibration data page initialization and switching
- Flash Programming for ECU development purposes
Table of contents

1. Architecture
2. Configuration
3. Integration and Runtime aspects
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. Debugging
   7. Measurement and Calibration
   8. Functional Safety
   9. Energy Management
Integration and Runtime aspects – Safety End to End (E2E) Communication Protection Overview

Typical sources of interferences, causing errors detected by E2E protection:

SW-related sources:
- S1. Error in mostly generated RTE, S2. Error in partially generated and partially hand-coded COM
- S3. Error in network stack
- S4. Error in generated IOC or OS

HW-related sources:
- H1. Microcontroller error during core/partition switch
- H2. Failure of HW network
- H2. Network EMI
- H3. Microcontroller failure during context switch (partition) or on the communication between cores

- AUTOSAR Confidential -

10 November 2011
Integration and Runtime aspects – Safety End to End (E2E) Communication Protection Logic

Notes:

- For each RTE Write or Read function that transmits safety-related data (like `Rte_Write_<p>_<o>()`), there is the corresponding E2E protection wrapper function.
- The wrapper function invokes AUTOSAR E2E Library.
- The wrapper function is a part of Software Component and is preferably generated.
- The wrapper function has the same signature as the corresponding RTE function, just instead of `Rte_` there is `E2EPW_`.
- The `E2EPW_` function is called by Application logic of SW-Cs, and the wrapper does the protection/checks and calls internally the RTE function.
- For inter-ECU communication, the data elements sent through E2E Protection wrapper are byte arrays. The byte arrays are put without any alterations in COM I-PDUs.
## Table of contents

1. Architecture
2. Configuration

3. Integration and Runtime aspects
   1. Mapping of Runnables
   2. Partitioning
   3. Scheduling
   4. Mode Management
   5. Error Handling, Reporting and Diagnostic
   6. Debugging
   7. Measurement and Calibration
   8. Functional Safety
   9. **Energy Management**
The goal of efficient energy management in AUTOSAR is to provide mechanisms for power saving, especially while bus communication is active (e.g. charging or clamp 15 active). AUTOSAR R3.2 and R4.0.3 support only Partial Networking.

Partial Networking

- Allows for turning off network communication across multiple ECUs in case their provided functions are not required under certain conditions. Other ECUs can continue to communicate on the same bus channel.
- Uses NM messages to communicate the request/release information of a partial network cluster between the participating ECUs.
Energy Management – Partial Networking
Example scenario of a partial network going to sleep

Initial situation:
- ECUs “A” and “B” are members of Partial Network Cluster (PNC) 1.
- ECUs “B”, “C” and “D” are members of PNC 2.
- All functions of the ECUs are organized either in PNC 1 or PNC 2.
- Both PNCs are active.
- PNC 2 is only requested by ECU “C”.
- The function requiring PNC 2 on ECU “C” is terminated, therefore ECU “C” can release PNC 2.

This is what happens:
- ECU “C” stops requesting PNC 2 to be active.
- ECU “C” and “D” are no longer participating in any PNC and can be shutdown.
- ECU “B” ceases transmission and reception of all signals associated with PNC 2.
- ECU “B” still participates in PNC 1. That means it remains awake and continues to transmit and receive all signals associated with PNC 1.
- ECU “A” is not affected at all.
A significant part of energy management is about mode handling. For the terms

- **Vehicle Mode**,  
- **Application Mode** and  
- **Basic Software Mode**

see chapter 3.4 of this document.

**Virtual Function Cluster (VFC):** groups the communication on port level between SW-components that are required to realize one or more vehicle functions.

This is the logical view and allows for a reusable bus/ECU independent design.

**VFC-Controller:** Special SW-component that decides if the functions of a VFC are required at a given time and requests or releases communication accordingly.

**Partial Network Cluster (PNC):** is a group of system signals necessary to support one or more vehicle functions that are distributed across multiple ECUs in the vehicle network.

This represents the system view of mapping a group of buses to one or more VFCs.
Energy Management – Partial Networking
Restrictions

- Partial Networking is currently supported on CAN and FlexRay buses.
- In order to wake-up a Partial Networking ECU a special transceiver hardware is required as specified in ISO 11898-5.
- The concept of Partial Networking does not consider a VFC located on only one ECU because there is no bus communication and no physical PNC.

- Restrictions for CAN
  - J1939 and Partial Networking exclude each other, due to the address claiming and start-up behaviour of J1939.
  - J1939 need to register first their address in the network before they are allowed to start communication after a wake-up.

- Restrictions on FlexRay
  - FlexRay is only supported for requesting and releasing PNCs.
  - FlexRay nodes cannot be shut down since there is no hardware available which supports partial networking.
Energy Management – Partial Networking
Mapping of Virtual Function Cluster to Partial Network Cluster

- SW-Component of VFC1
- SW-Component of VFC2
- SW-Component of VFC3
- Communication Port

Mapping of VFC on PNC

- Here both Partial Networks map to one CAN bus.
- One Partial Network can also span more than one bus.
Energy Management – Partial Networking
Involved modules – Solution for CAN

- AUTOSAR Confidential -