Document Title	Layered Software Architecture
Document Category	Explanation (EXP)
Document UID	53
Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R26-11

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2026-11-26	R26-11	AUTOSAR Release Management	➢ Removed several modules: HWTestManager, V2X modules (including their drivers), FlexRayARTransportLayer ➢ Smaller clarifications and corrections (terminology, updates to diagrams)
2025-11-27	R25-11	AUTOSAR Release Management	➢ Added information about VDP ➢ Added information about Mirror ➢ Updated slides about libraries, DDS ➢ Removed TTCan ➢ Removed Fls and Eep ➢ Removed LdCom ➢ Minor changes
2024-11-27	R24-11	AUTOSAR Release Management	➢ Added L-SDU Router ➢ Incorporated J1939Fscp transformer into comm stack extensions ➢ Incorporated Partitioning examples ➢ Incorporated migration of BSWModuleList to BSWGeneral ➢ Added I2C in Comm Drivers ➢ Improve documentation about error handling and usage ➢ Cleanup remains of partition restart ➢ Added changes in memory manipulation library: Copy, Set, Move, Compare ➢ Remove E2EPW support ➢ Added changes to error handling

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2023-11-23	R23-11	AUTOSAR Release Management	➢ Added information about charging management (ChrgM) and firewall ➢ Editorial changes
2022-11-24	R22-11	AUTOSAR Release Management	➢ Incorporated new concepts for Vehicle-2-X Data Manager, MACsec, CAN XL, DDS, Secured Time Synchronization, Vehicle-2-X Support for China ➢ Editorial changes
2021-11-25	R21-11	AUTOSAR Release Management	➢ Incorporated draft concept for new Memory Driver and Memory Access
2020-11-30	R20-11	AUTOSAR Release Management	➢ Removed Pretended Networking ➢ Added caveats for E2E Protection Wrapper ➢ Layer Interaction Matrix: Allow Crypto Driver to access Memory Services ➢ Incorporated new concepts for Intrusion Detection System Manager, CP Software Clusters
2019-11-28	R19-11	AUTOSAR Release Management	➢ Incorporated new concepts for Atomic multicore safe operations, Signal-service-translation, NV data handling enhancement ➢ Changed Document Status from Final to published

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2018-10-31	4.4.0	AUTOSAR Release Management	➢ Adopting LIN Slave Support, LinNm removed ➢ New Concepts: Key Management, 1st draft of MCAL Multicore Distribution ➢ Editorial changes
2017-12-08	4.3.1	AUTOSAR Release Management	➢ Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	➢ Incorporated new 4.3 concepts for Crypto Stack, Vehicle-2-X Communication, SOME/IP Transport Protocol, DLT rework ➢ Removed obsolete Dbg module ➢ Editorial changes
2015-07-31	4.2.2	AUTOSAR Release Management	➢ Editorial changes

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2014-10-31	4.2.1	AUTOSAR Release Management	➢ Incorporated new 4.2 concepts for: Switch Configuration; Sender-Receiver-Serialization; CAN-FD; Large-Data-COM; E2E-Extension; Global Time Synchronization; Support for Post-build ECU-Configuration; Secure-Onboard-Communication; ASIL/QM-Protection ➢ Introduction of new error classification ➢ Editorial changes
2014-03-31	4.1.3	AUTOSAR Release Management	➢ Editorial changes
2013-03-15	4.1.1	AUTOSAR Administration	➢ Clarification of partial network support for CAN/LIN slave. ➢ New Ethernet stack extensions ➢ Added Crypto Service Manager to System Services ➢ Revised presentation of J1939 and added new J1939 modules ➢ Added new energy management concepts: “Pretended Networking”, “ECU Degradation” ➢ Added new modules: “Output Compare Unit Driver” and “Time Service” ➢ Changed handling of Production Errors ➢ Fixed various typography and layout issues
2011-12-22	4.0.3	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“

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2009-12-18	4.0.1	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"
2010-02-02	3.1.4	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
2008-08-13	3.1.1	AUTOSAR Administration	➢ Legal disclaimer revised
2007-12-21	3.0.1	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

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2007-01-24	2.1.15	AUTOSAR Administration	➢ ICC clustering added. ➢ Document contents harmonized ➢ Legal disclaimer revised ➢ Release Notes added ➢ “Advice for users” revised ➢ “Revision Information” added
2006-11-28	2.1.1	AUTOSAR Administration	Rework Of: ➢ Error Handling ➢ Scheduling Mechanisms ➢ More updates according to architectural decisions in R2.0
2006-01-02	1.0.1	AUTOSAR Administration	➢ Correct version released
2005-05-31	1.0.0	AUTOSAR Administration	➢ Initial release

Disclaimer

Disclaimer

This work (specification and/or software implementation) 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 work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work 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 work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

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. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. Interfaces: General Rules
    7. Interfaces: Interaction of Layers
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Purpose of this document

The **Layered Software Architecture** describes the software architecture of AUTOSAR:

- it describes in an 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.

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

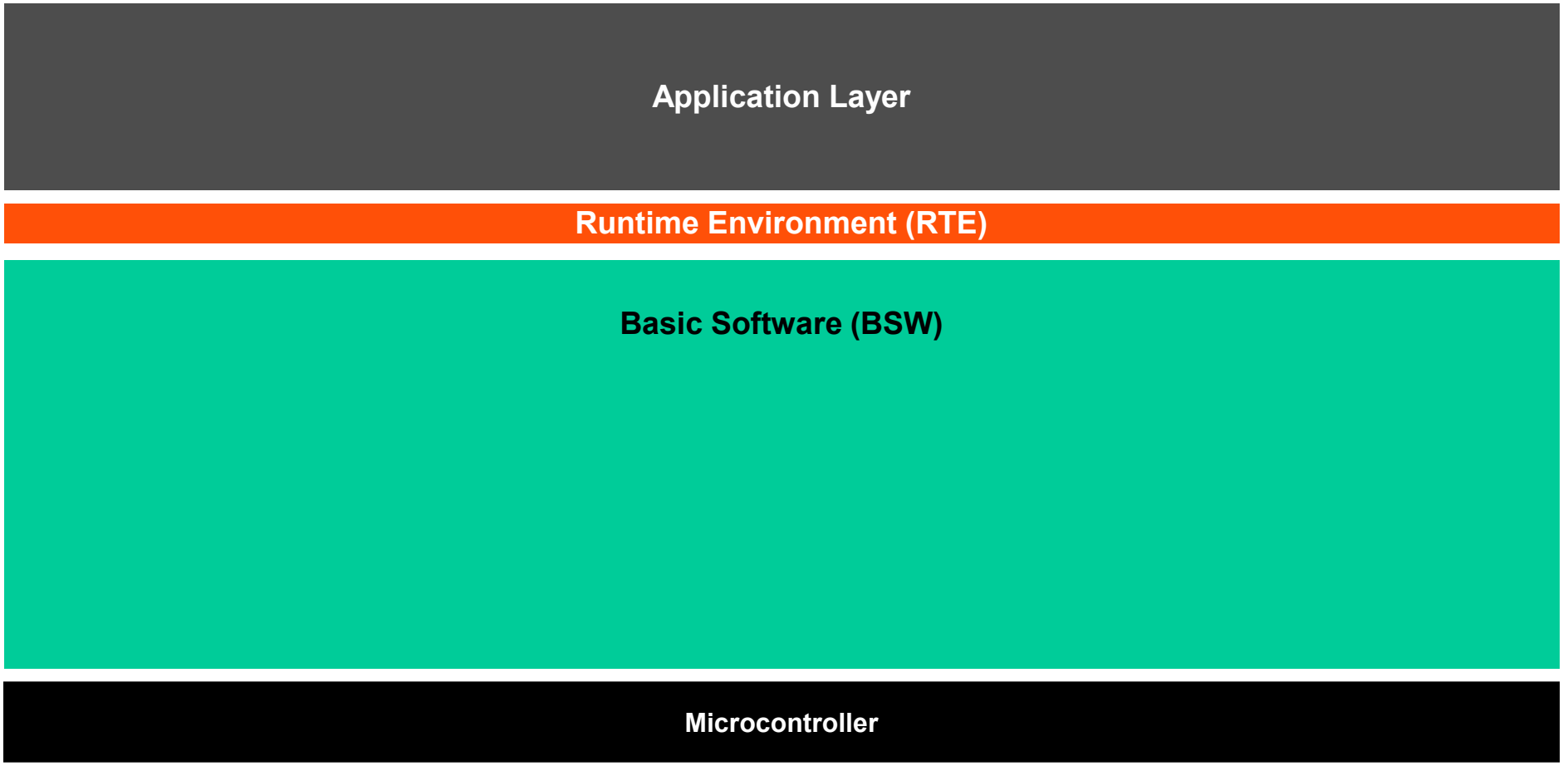
The AUTOSAR Software Architecture is a generic approach:

- 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.

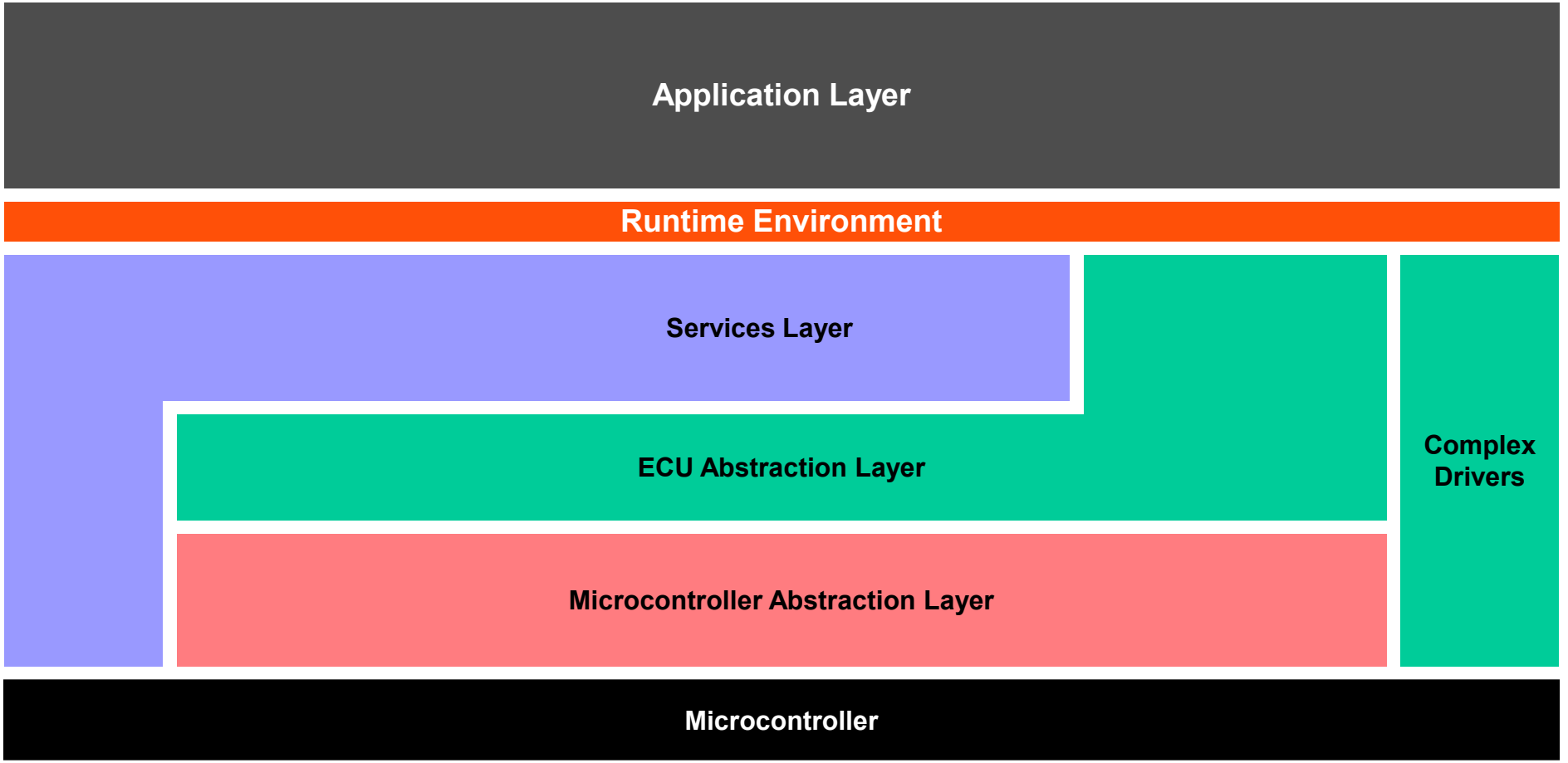
Architecture – Overview of Software Layers
Top view

The AUTOSAR Architecture distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller.



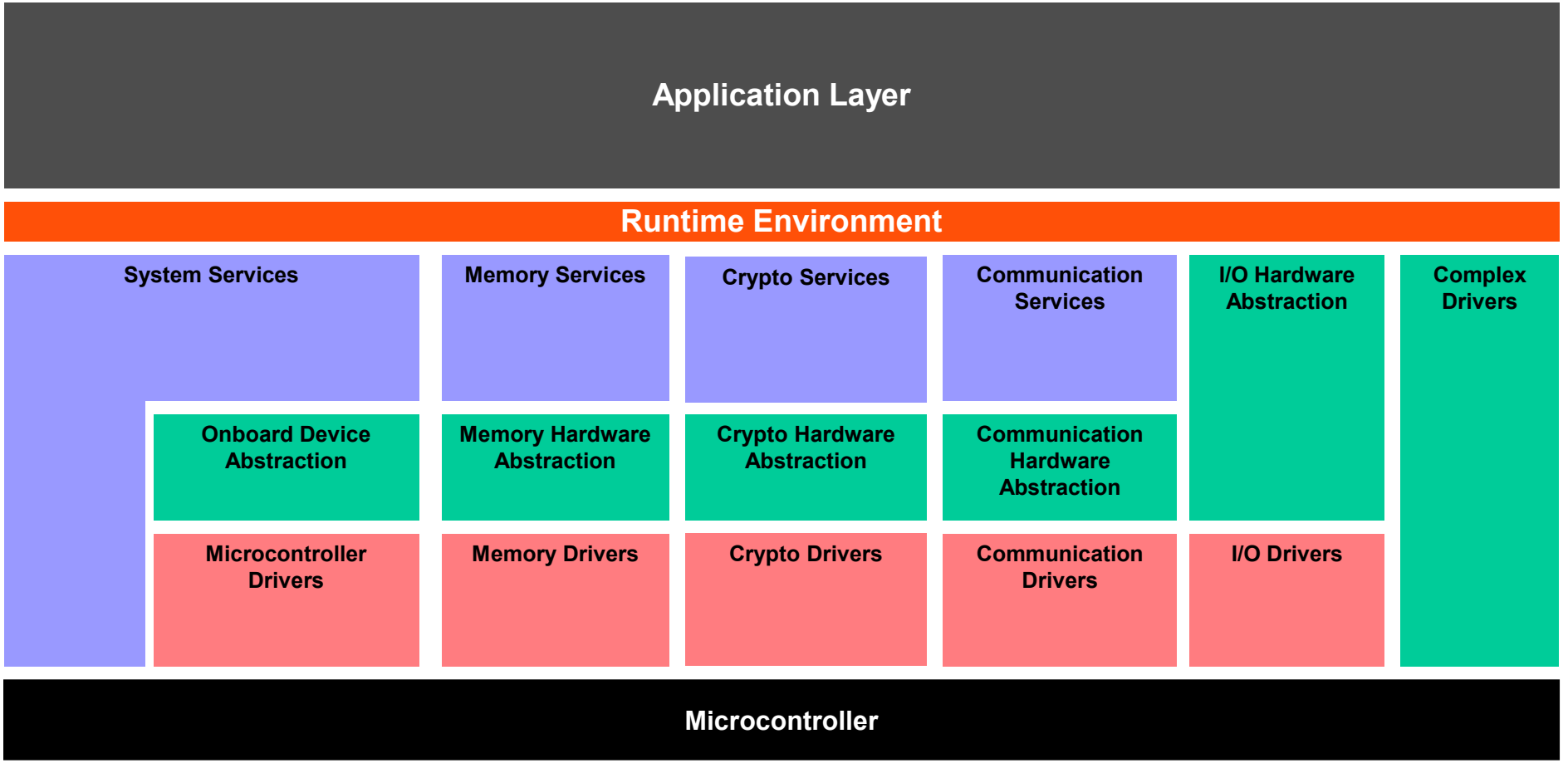
Architecture – Overview of Software Layers
Coarse view

The AUTOSAR Basic Software is further divided in the layers: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers.



Architecture – Overview of Software Layers
Detailed view

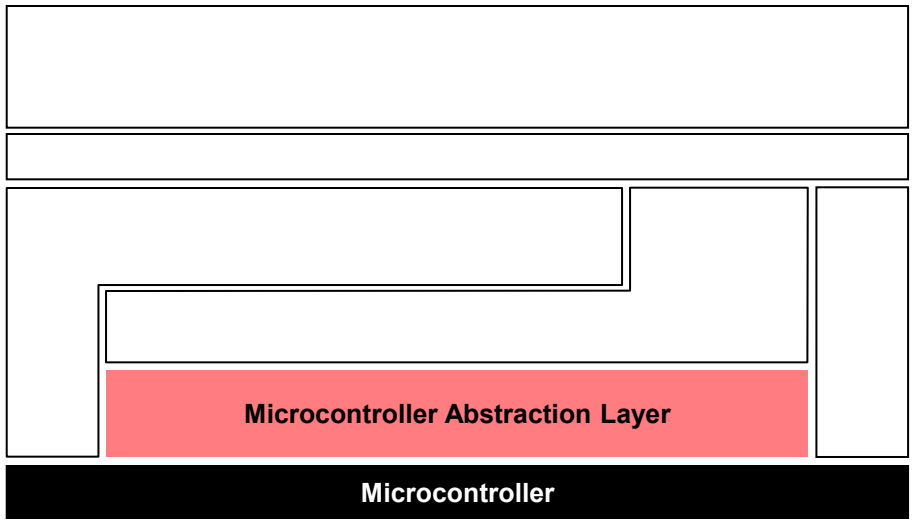
The Basic Software Layers are further divided into functional groups. Examples of Services are System, Memory and Communication Services.



Architecture – Overview of Software Layers
Microcontroller Abstraction Layer

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

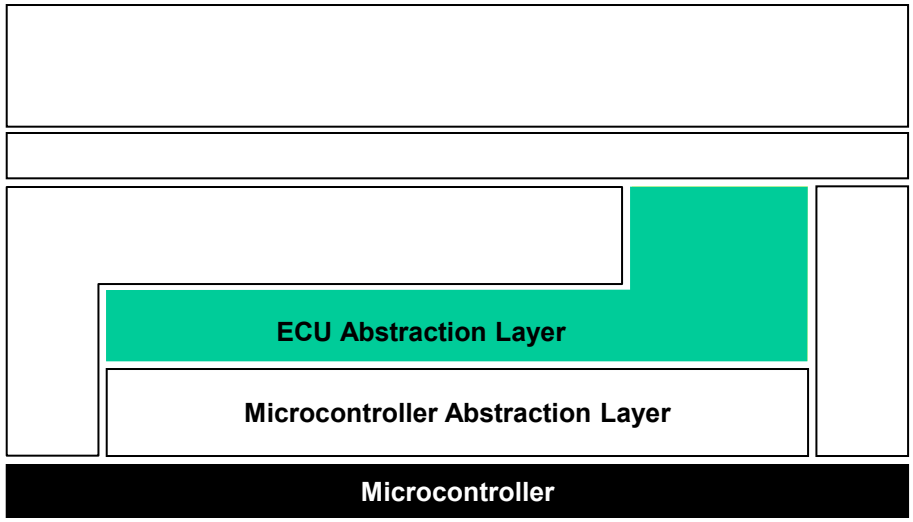
Implementation: µC dependent

Upper Interface: standardized and µC independent

Architecture – Overview of Software Layers
ECU Abstraction Layer

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

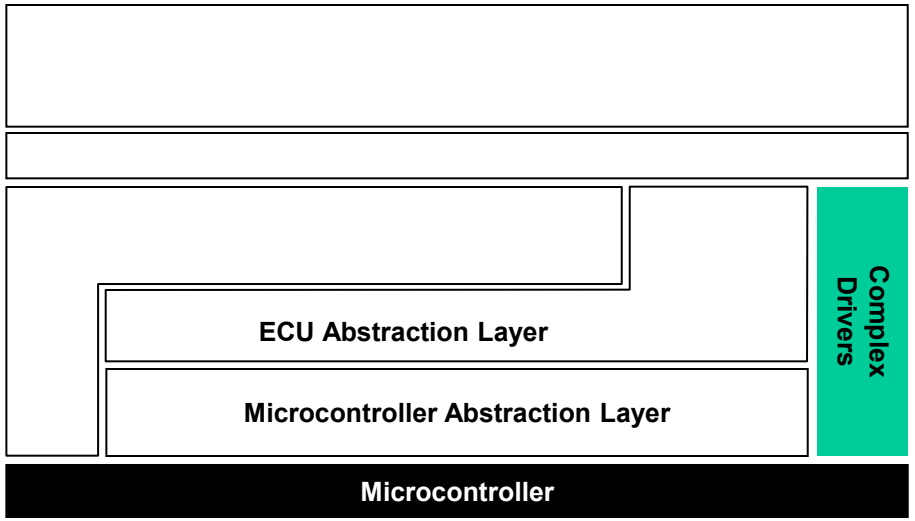
Properties

Implementation: µC independent, ECU hardware dependent

Upper Interface: µC and ECU hardware independent

Architecture – Overview of Software Layers
Complex Drivers

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 constrains 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

Architecture – Overview of Software Layers
Services Layer

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 (WdgManager)

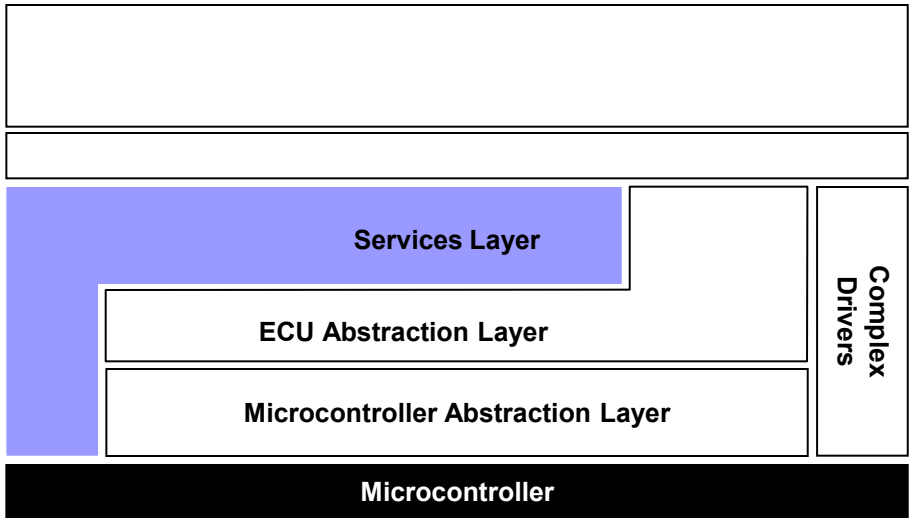
Task

Provide basic services for applications, RTE and basic software modules.

Properties

Implementation: mostly µC and ECU hardware independent

Upper Interface: µC and ECU hardware independent

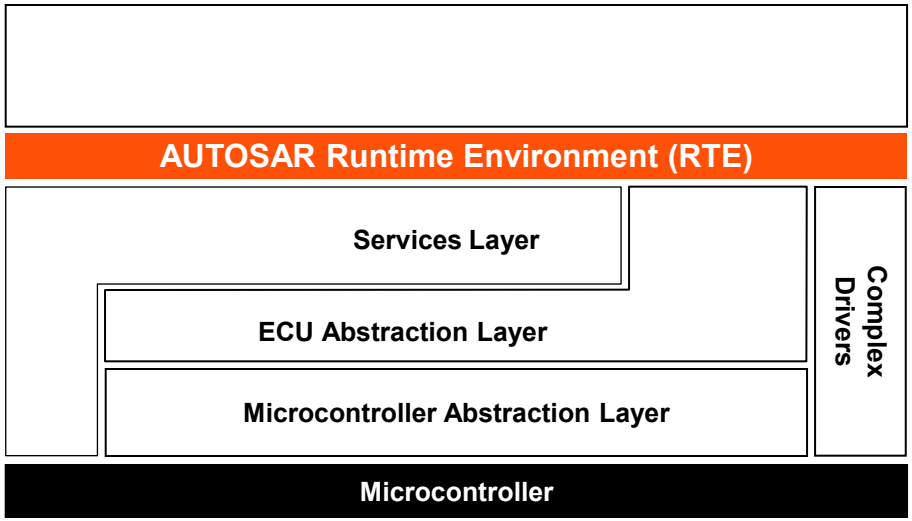


Architecture – Overview of Software Layers
AUTOSAR Runtime Environment (RTE)

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

Introduction to types of services

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)
- **Crypto**
  Standardized access to cryptographic primitives including internal/external hardware accelerators
- **Communication**
  Standardized access to: vehicle network systems, ECU onboard communication systems and ECU internal SW
- **System**
  Provision of standardizable (operating system, timers, error memory) and ECU specific (ECU state management, watchdog manager) services and library functions

Driver (internal)

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.

Interface

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).

Handler

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

The handler does not change the content of the data.

Handler functionality is often incorporated in the driver or interface (e.g. SPIHandlerDriver, ADC Driver).

Manager

A **manager** offers specific services for multiple clients. It is needed in all cases where pure handler functionality is not enough 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**

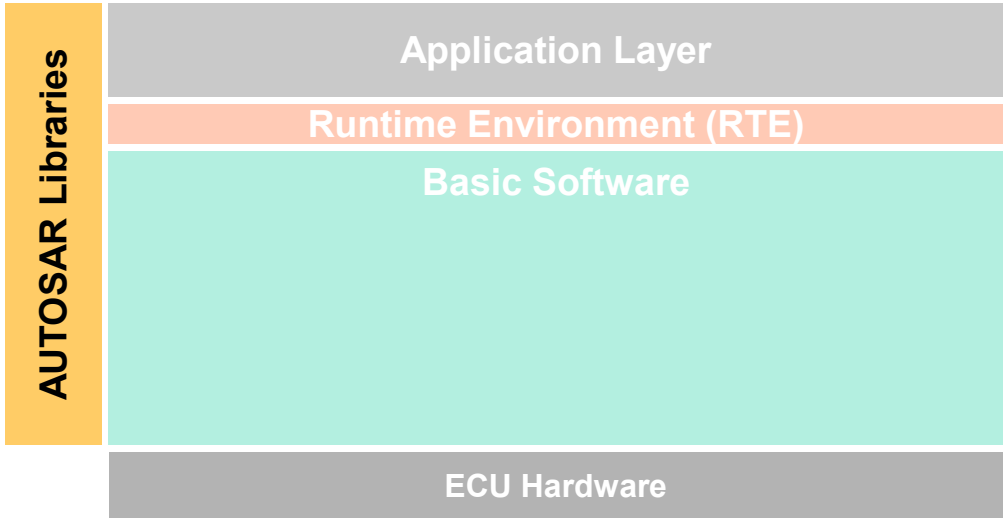
Example: The NVRAM manager manages the concurrent access to internal and/or external memory devices like flash and EEPROM memory. It also performs distributed and reliable data storage, data checking, provision of default values etc.

Architecture – Overview of Software Layers
Introduction to Libraries (1)

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, no calls to BSW modules are allowed
- are re-entrant
- do not have internal states
- do not require any initialization
- are synchronous, i.e. they do not have wait points
- may have a configuration, but this is discouraged



Introduction to Libraries (2)

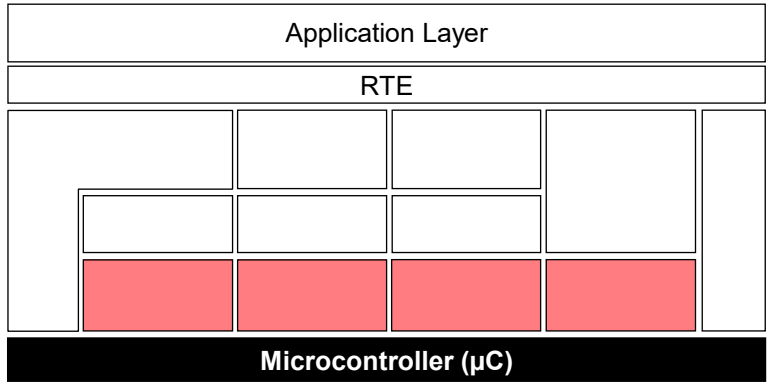
The following libraries are specified within AUTOSAR:

- Fixed point mathematical (Mfx),
- Floating point mathematical (Mfl),
- Interpolation for fixed point data Ifx),
- Interpolation for floating point data (Ifl),
- Extended functions (e.g. 64bits calculation, filtering, etc.) (Efx),
- Bit handling (Bfx),
- E2E communication (E2E),
- CRC calculation (Crc),
- Atomic multicore safe operations (Bmc),
- Memory standard functions (Msf)

Table of contents

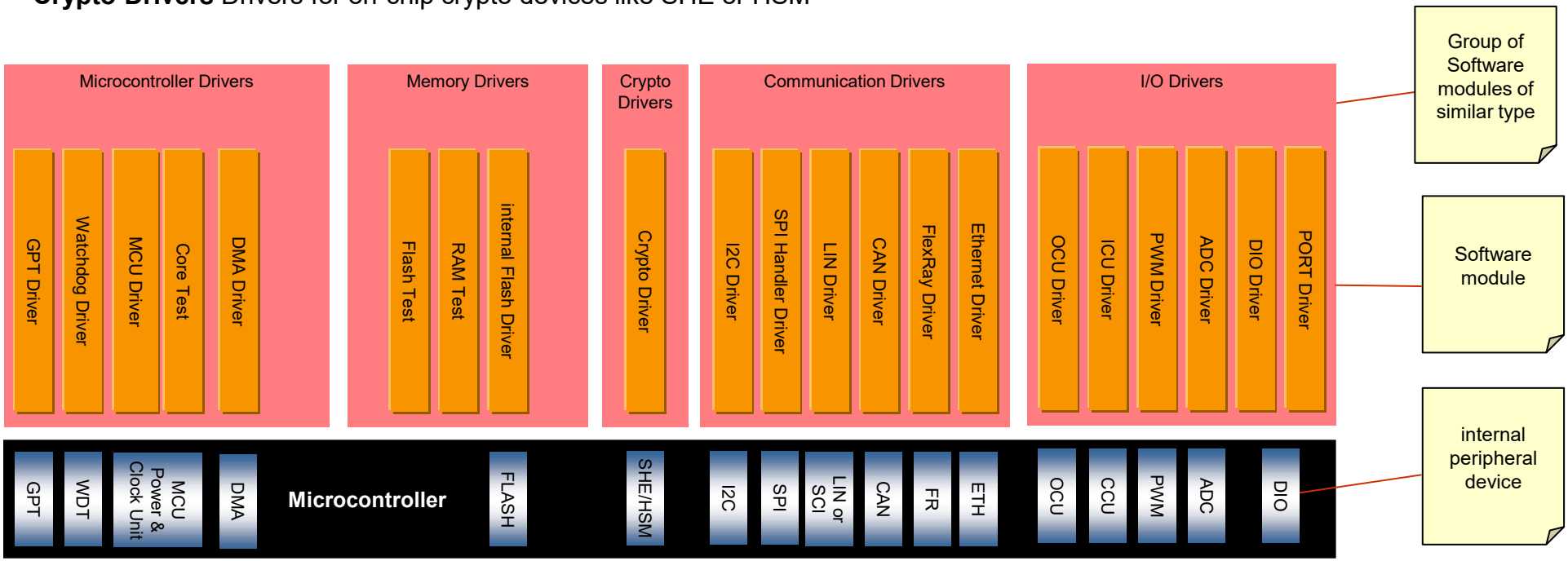
1. **Architecture**
    1. Overview of Software Layers
    2. **Content of Software Layers**
    3. Content of Software Layers in Multi-Core Systems
    4. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. Interfaces: General Rules
    7. Interfaces: Interaction of Layers
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Architecture – Content of Software Layers
Microcontroller Abstraction Layer

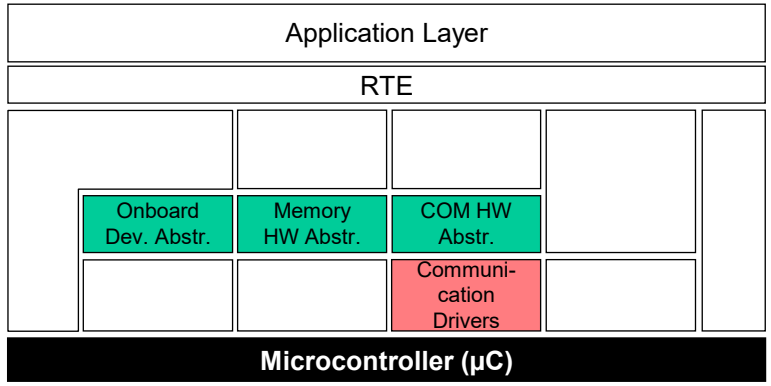


The **µC Abstraction Layer** consists of the following module groups:

- **Microcontroller Drivers**
  Drivers for internal peripherals (e.g. Watchdog, General Purpose Timer)
  Functions with direct µC access (e.g. Core test)
- **Communication Drivers**
  Drivers for ECU onboard (e.g. SPI) and vehicle communication (e.g. CAN).
  OSI-Layer: Part of Data Link Layer
- **Memory Drivers**
  Drivers for on-chip memory devices (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash)
- **I/O Drivers:**
  Drivers for analog and digital I/O (e.g. ADC, PWM, DIO)
- **Crypto Drivers** Drivers for on-chip crypto devices like SHE or HSM



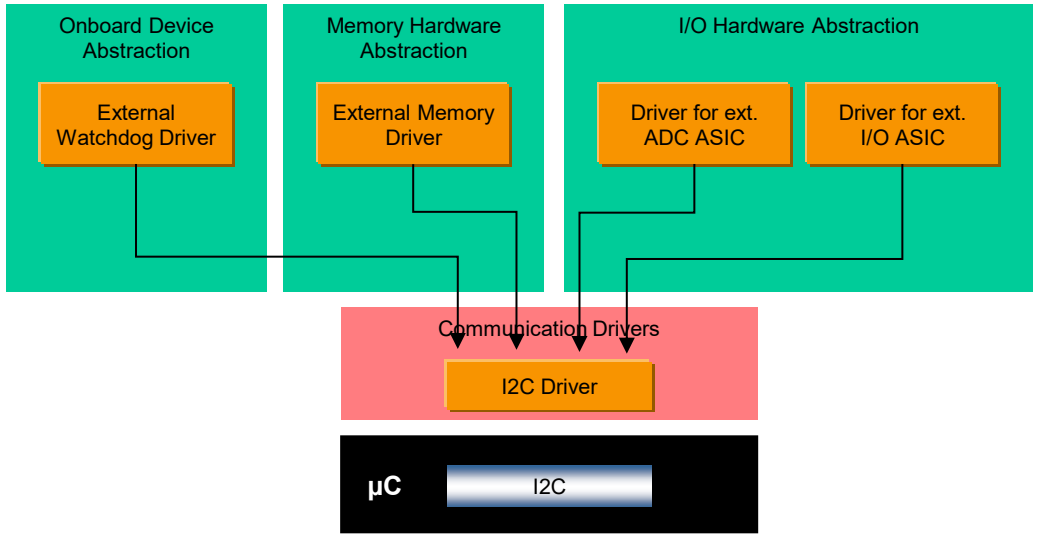
Architecture – Content of Software Layers
Microcontroller Abstraction Layer: I2C Driver



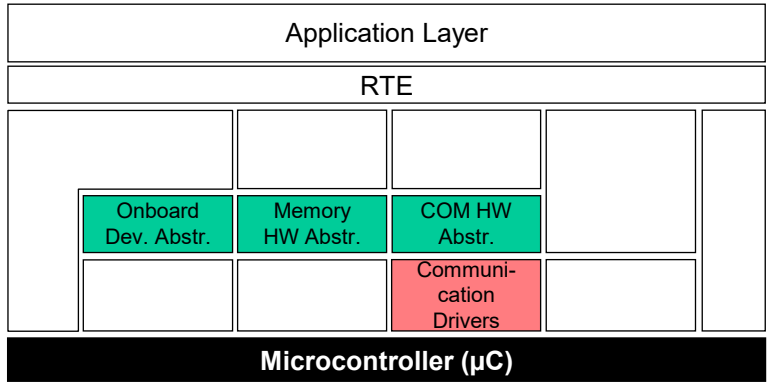
The **I2C Driver** allows concurrent access of several nodes to one or more I2C busses.

- I²C (Inter-Integrated Circuit) is a 2-wire serial data bus widely used in automotive sensors or actuators.
- Examples:
  - Temperature Sensor
  - 3-axis Accelerometer Sensor
  - Air Pressure Sensor
  - EEPROM
  - …

Example:



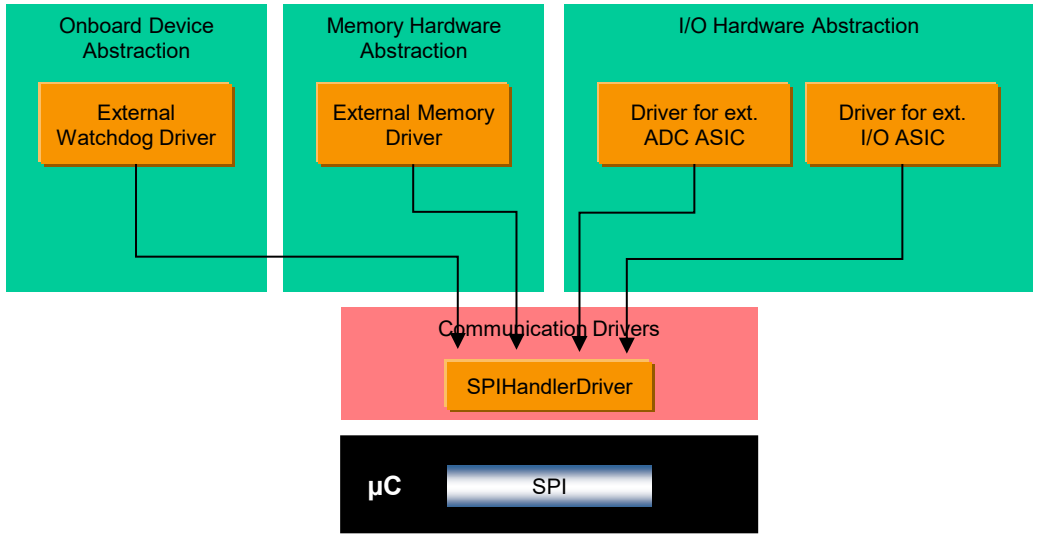
Architecture – Content of Software Layers
Microcontroller Abstraction Layer: SPIHandlerDriver



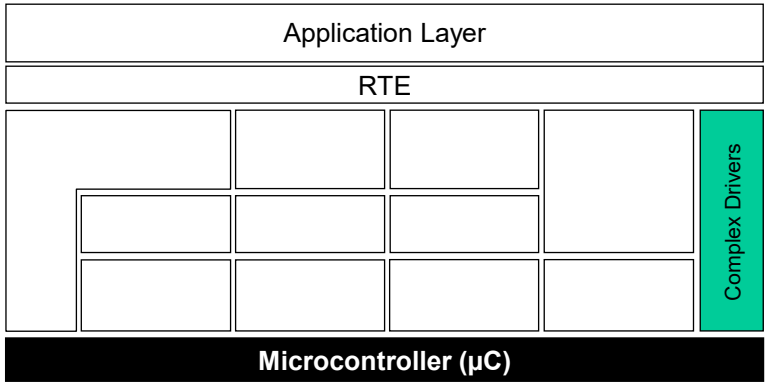
The **SPIHandlerDriver** allows concurrent access of several clients to one or more SPI busses.

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:



Architecture – Content of Software Layers
Complex Drivers



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

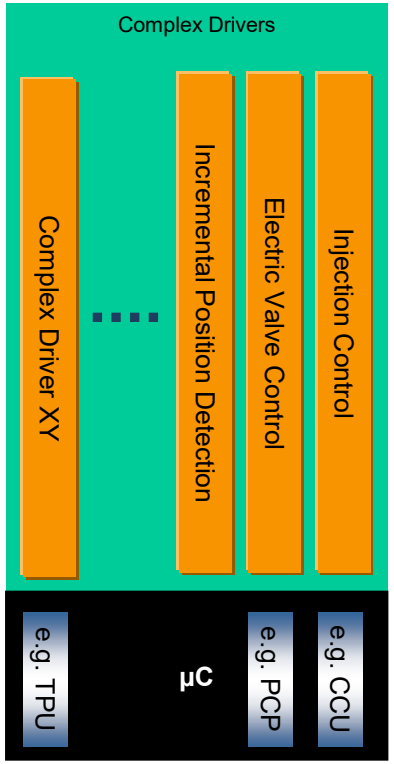
Properties:

Implementation: highly µC, ECU and application dependent

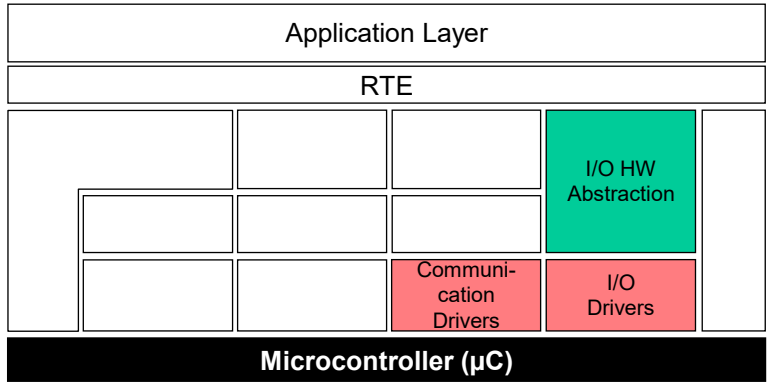
Upper Interface to SW-Cs: specified and implemented according to AUTOSAR (AUTOSAR interface)

Lower interface: restricted access to Standardized Interfaces

Example:



Architecture – Content of Software Layers
ECU Abstraction*: I/O Hardware Abstraction*



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

The different I/O devices 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).

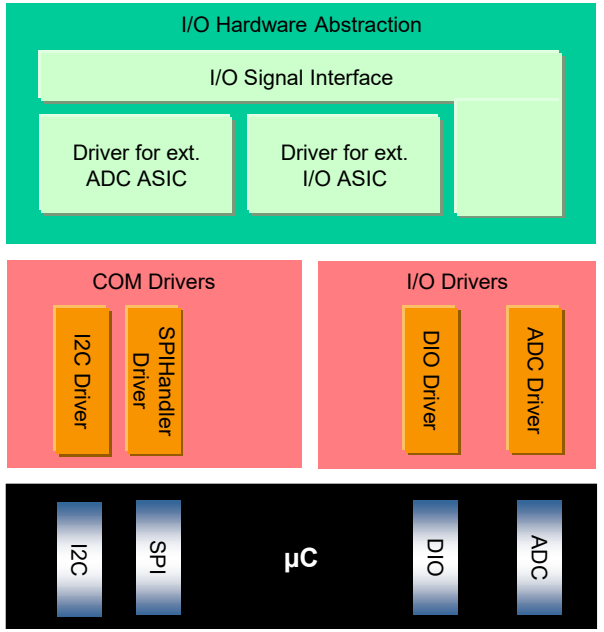
Hide ECU hardware and layout properties from higher software layers.

Properties:

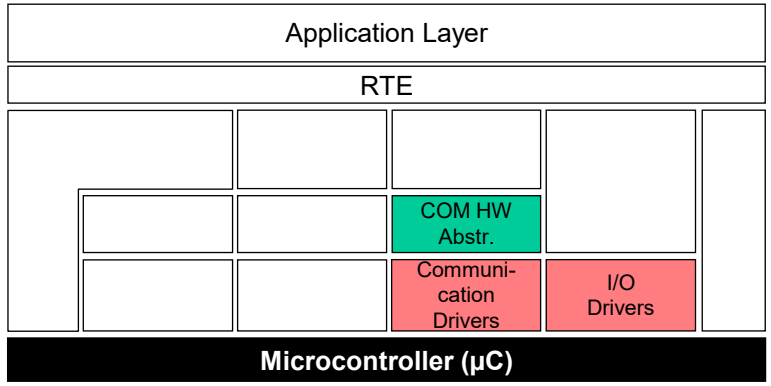
Implementation: µC independent, ECU hardware dependent

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

Example:



Architecture – Content of Software Layers
ECU Abstraction*: Communication Hardware Abstraction*

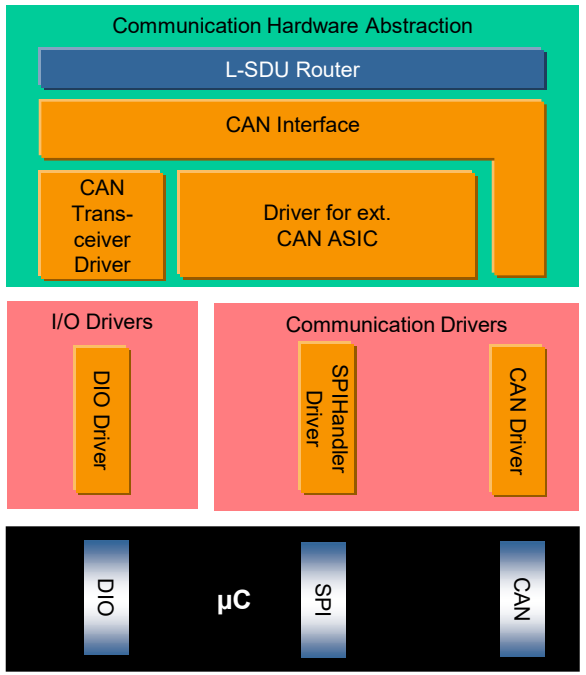


The **Communication Hardware Abstraction** is a group of modules which abstracts from the ***location*** of communication controllers and the ***ECU hardware layout***. For all communication systems a ***specific*** Communication Hardware Abstraction is required (e.g. for LIN, CAN, FlexRay).

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).

Example:



Task:

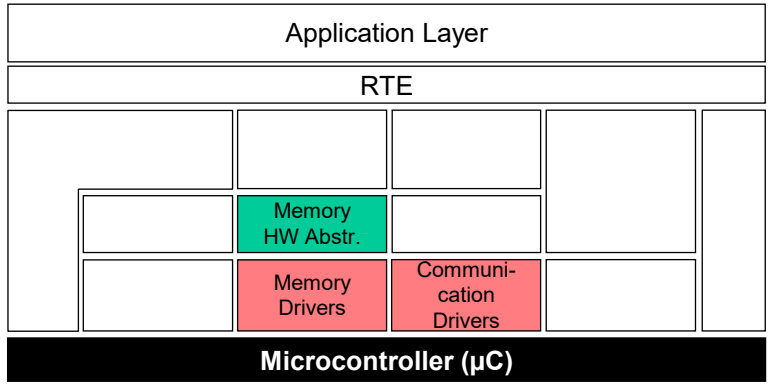
Provide equal mechanisms to access a bus channel regardless of it‘s location (on-chip / on-board)

Properties:

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

Upper Interface: bus dependent, µC and ECU hardware independent

Architecture – Content of Software Layers
Scope: Memory Hardware Abstraction



The **Memory Hardware Abstraction** is a group of modules which abstracts from the ***location*** of peripheral memory devices (on-chip or on-board) and the ***ECU hardware layout***.

Example: on-chip EEPROM and external EEPROM devices 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:

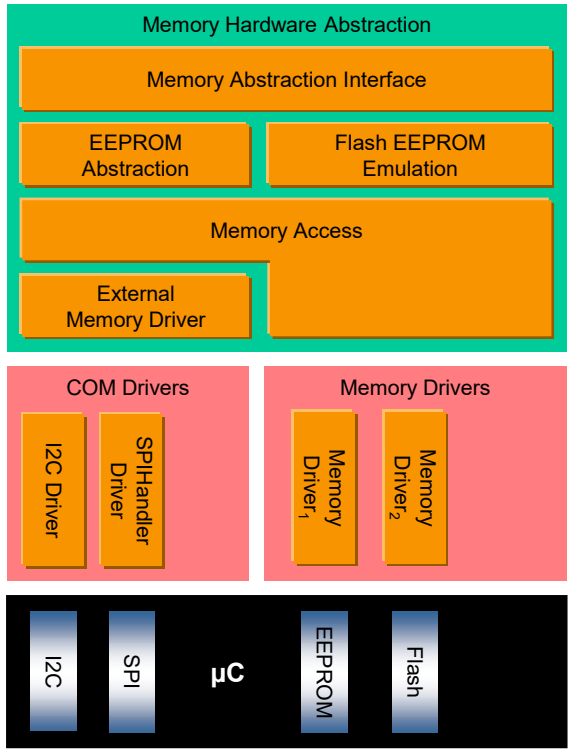
Provide equal mechanisms to access internal (on-chip) and external (on-board) memory devices and type of memory hardware (EEPROM, Flash).

Properties:

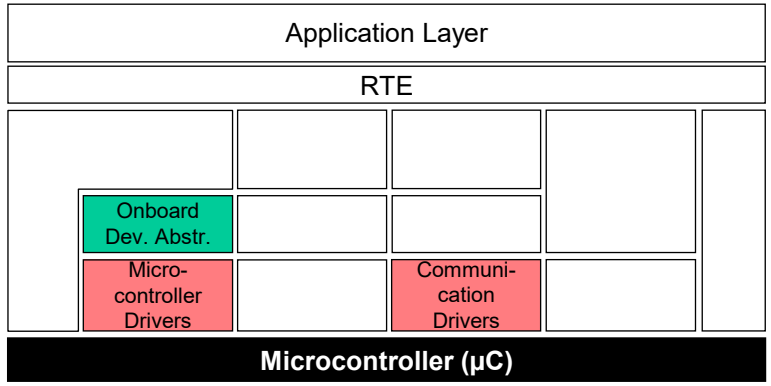
Implementation: µC independent, external device dependent

Upper Interface: µC, ECU hardware and memory device independent

Example:



Architecture – Content of Software Layers
Onboard Device Abstraction

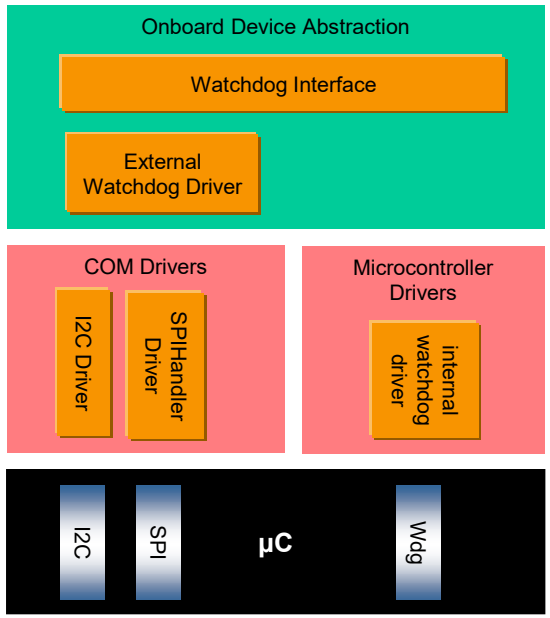


The **Onboard Device Abstraction** contains drivers for ECU onboard devices which cannot be seen as sensors or actuators like internal or external watchdogs. Those drivers access the ECU onboard devices via the µC Abstraction Layer.

Task:

Abstract from ECU specific onboard devices.

Example:

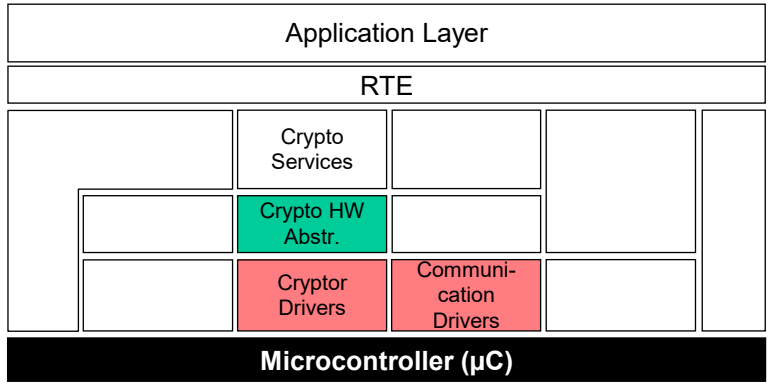


Properties:

Implementation: µC independent, external device dependent

Upper Interface: µC independent, partly ECU hardware dependent

Architecture – Content of Software Layers
Scope: Crypto Hardware Abstraction



The **Crypto Hardware Abstraction** is a group of modules which abstracts from the ***location*** of cryptographic primitives (internal- or external hardware or software-based).

Example: AES primitive is realized in SHE or provided as software library

Task:

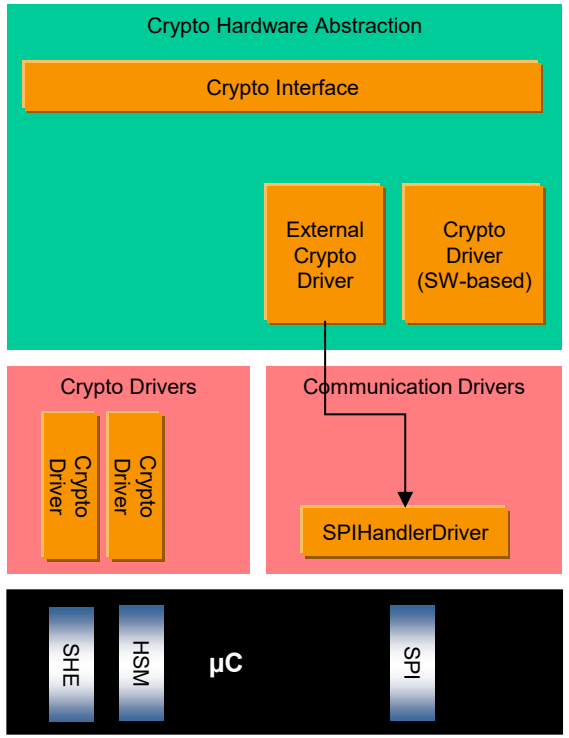
Provide equal mechanisms to access internal (on-chip) and software cryptographic devices.

Properties:

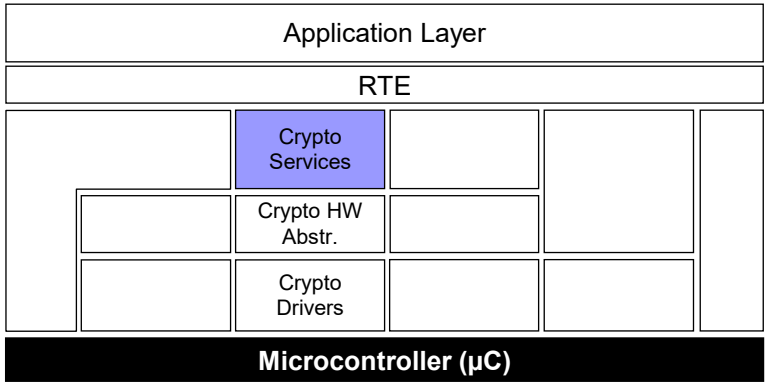
Implementation: µC independent

Upper Interface: µC, ECU hardware and crypto device independent

Example:



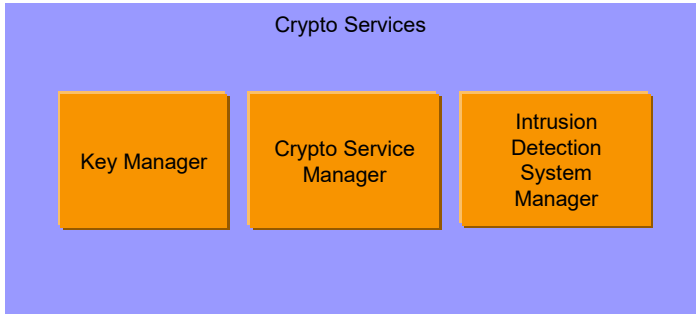
Architecture – Content of Software Layers
Services: Crypto Services



The **Crypto Services** consist of three modules

- the Crypto Service Manager is responsible for the management of cryptographic jobs
- the Key Manager interacts with the key provisioning master (either in NVM or Crypto Driver) and manages the storage and verification of certificate chains
- The Intrusion Detection System Manager is responsible for handling security events reported by BSW modules or SW-C

Example:



Task:

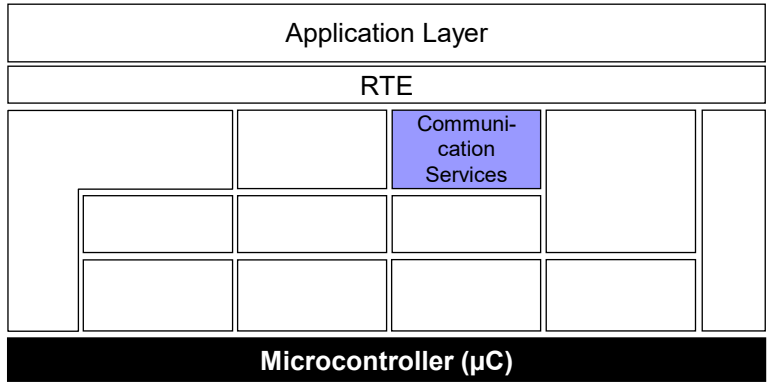
Provide cryptographic primitives, IDS services and key storage to the application in a uniform way. Abstract from hardware devices and properties.

Properties:

Implementation: µC and ECU hardware independent, highly configurable

Upper Interface: µC and ECU hardware independent specified and implemented according to AUTOSAR (AUTOSAR interface)

Architecture – Content of Software Layers
Communication Services – General



The **Communication Services** are a group of modules for vehicle network communication (CAN, LIN, FlexRay and Ethernet). 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.

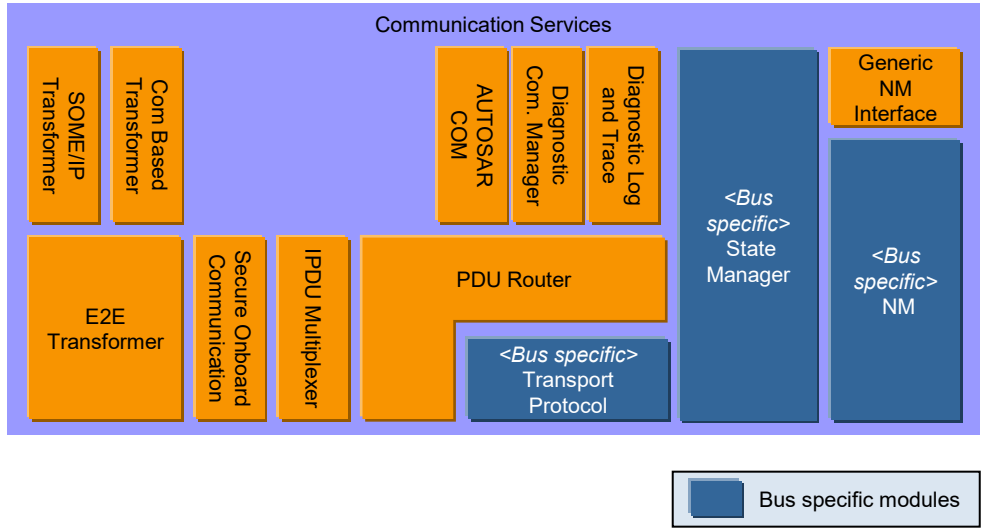
Properties:

Implementation: µC and ECU HW independent, partly dependent on bus type

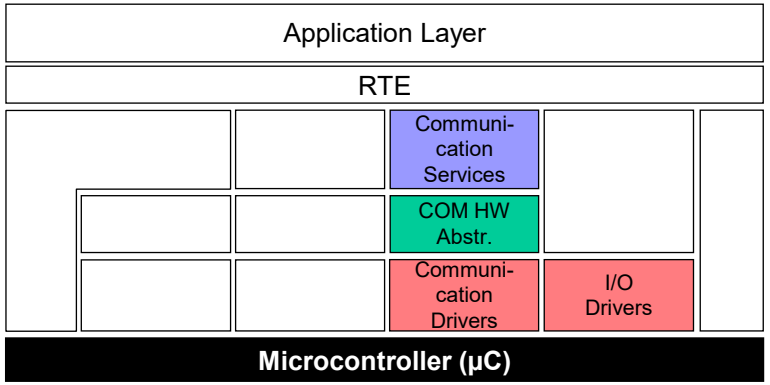
Upper Interface: µC, ECU hardware and bus type independent

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

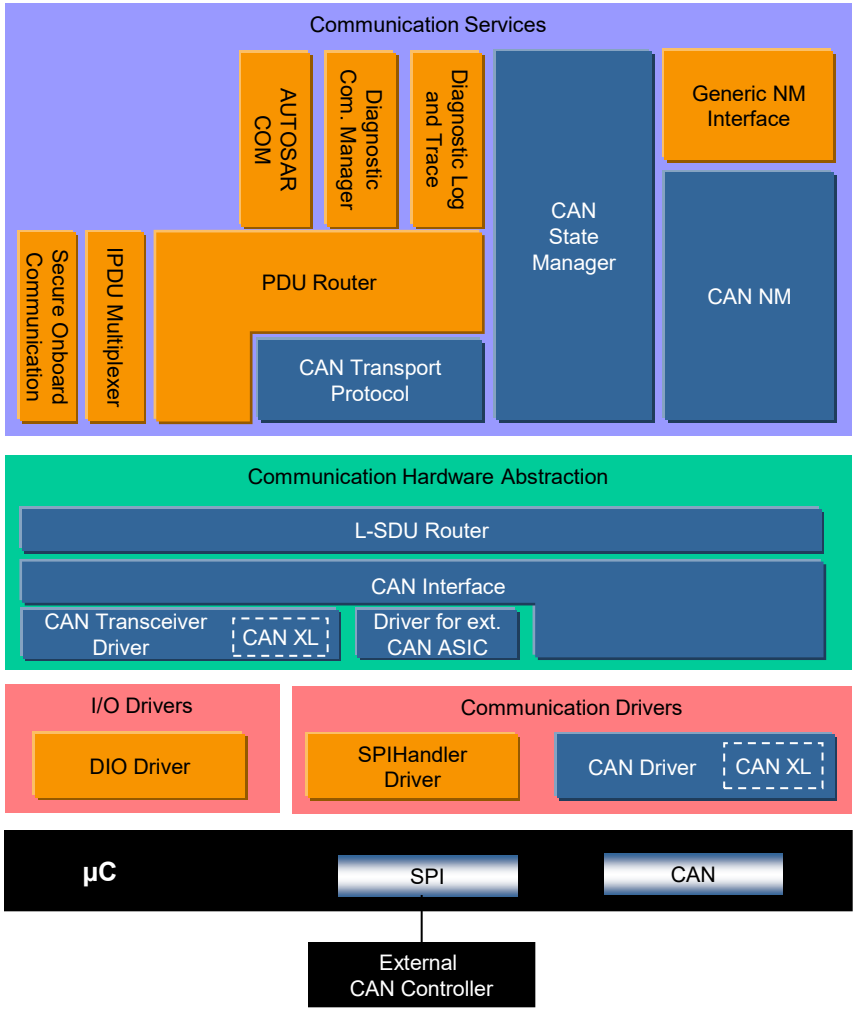
Example:



Architecture – Content of Software Layers
Communication Stack – CAN



Example:



The **CAN Communication Services** are a group of modules for vehicle network communication with the communication system CAN.

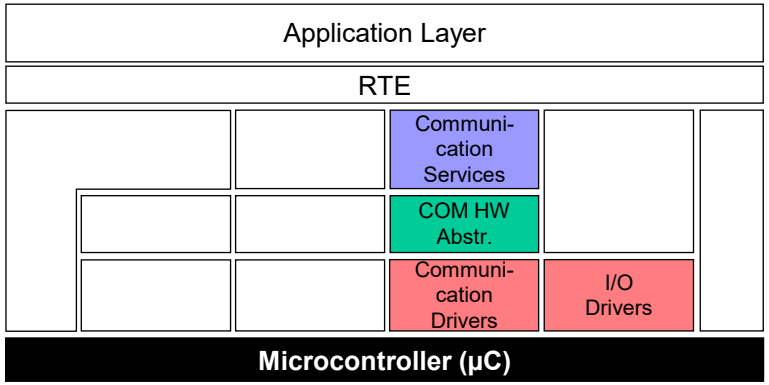
Task:

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

The **CAN Communication Stack** supports:

- Classic CAN communication (CAN 2.0)
- CAN FD communication, if supported by hardware
- CAN XL communication, if supported by hardware

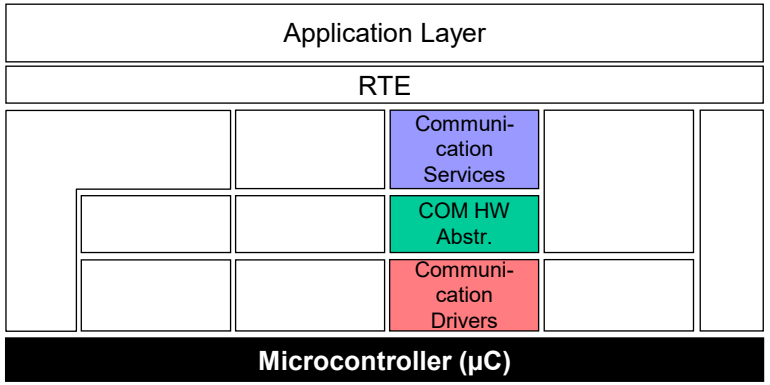
Architecture – Content of Software Layers
Communication Stack – CAN



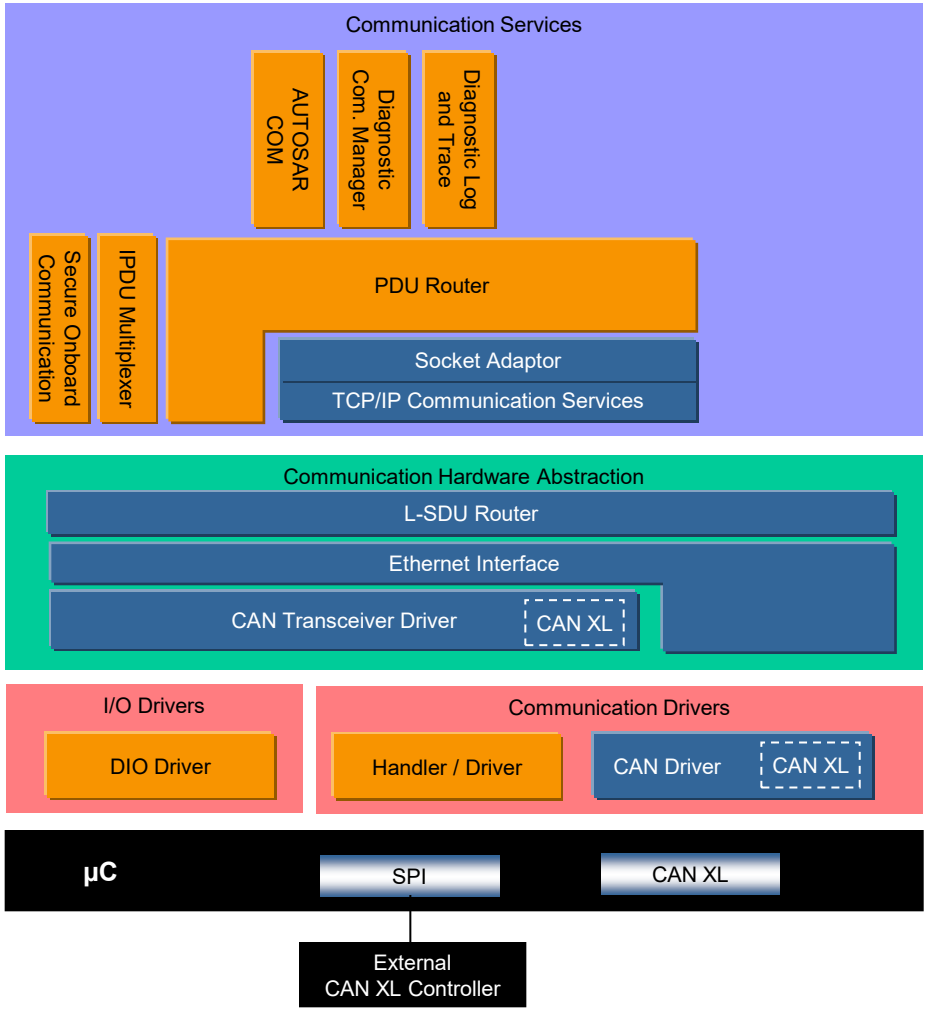
Properties:

- 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 NM is specific for CAN networks and will be instantiated per CAN vehicle network system.
- 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.

Architecture – Content of Software Layers
Communication Stack – Ethernet/CAN XL



Example:

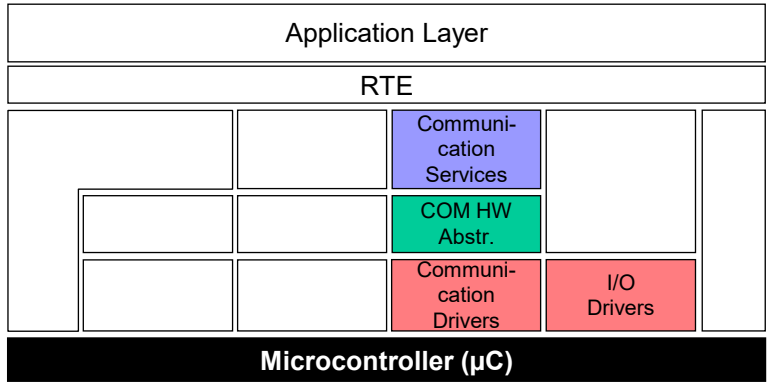


CAN XL supports to directly tunnel **IEEE 802.3 Ethernet frames** for participation of IP communication.

Task:

- Provide vehicle wide communication with same semantic used everywhere regardless physical connection (CAN XL / Ethernet) or communication paradigm (Signal- and Service-based communication).

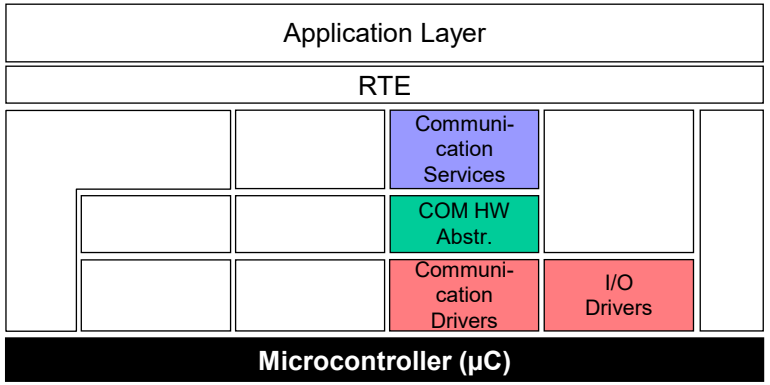
Architecture – Content of Software Layers
Communication Stack Extension – CAN XL



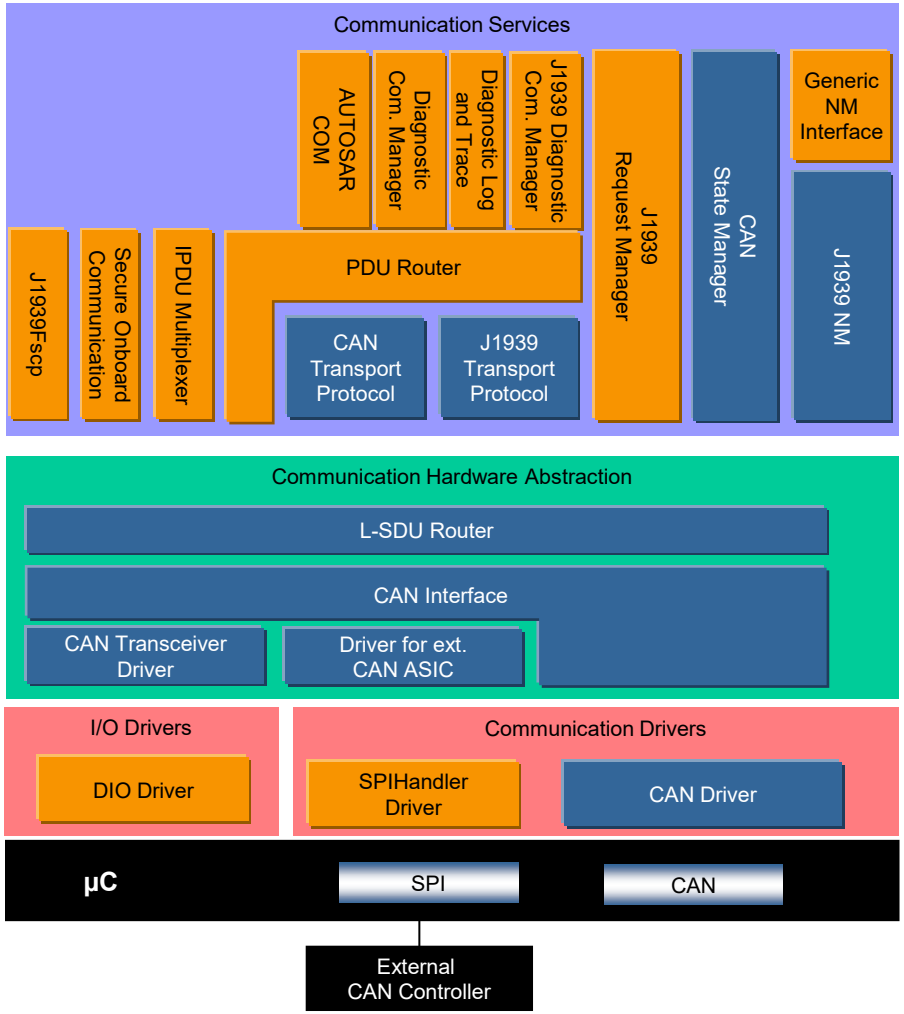
Properties:

- CAN XL is an absolute superset to CAN, i.e. a CAN stack which supports CAN XL can serve both a CAN and a CAN XL bus.
- CanIf, CanTrcvDrv and CanDrv are the only modules which need extensions to serve CAN XL communication.
- The properties of the communication stack CAN are also true for CAN with CAN XL functionality.

Architecture – Content of Software Layers
Communication Stack Extension – J1939



Example:



The **J1939 Communication Services** extend the plain CAN communication stack for vehicle network communication in heavy duty vehicles.

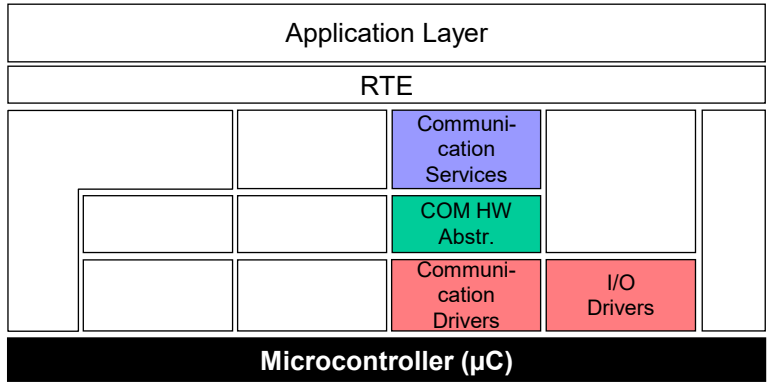
Task:

- Provide the protocol services required by J1939. Hide protocol and message properties from the application where not required.

Please Note:

- There are two transport protocol modules in the CAN stack (CanTp and J1939Tp) which can be used alternatively or in parallel on different channels:. 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

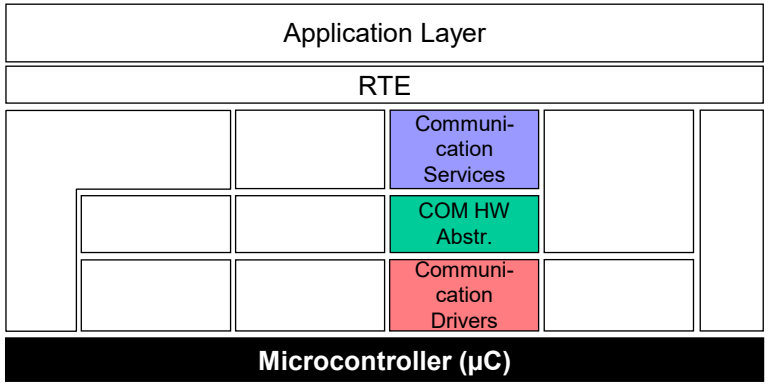
Architecture – Content of Software Layers
Communication Stack Extension – J1939



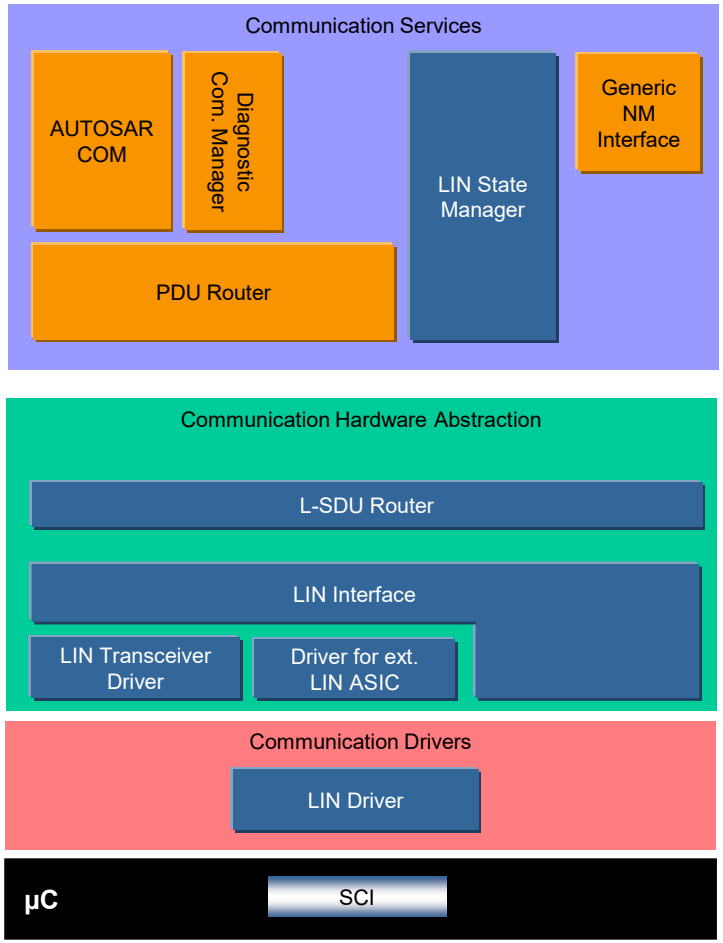
Properties:

- Implementation: µC and ECU HW independent, based 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.
- Supports dynamic frame identifiers that are not known at configuration time.
- J1939 network management handles assignment of unique addresses to each ECU but does not support sleep/wakeup handling and related concepts like partial networking.
- Provides J1939 diagnostics and request handling.
- Provides support for Functional Safety per SAE J1939-76, including initialization, Safety Data Group (SDG) Operations, Safety Header/Data Message processing using PROFILE_J1939_76 E2E Transformer, and Error Management between producer and consumer Application SWCs.

Architecture – Content of Software Layers
Communication Stack – LIN



Example:



The **LIN Communication Services** are a group of modules for vehicle network communication with the communication system LIN.

Task:

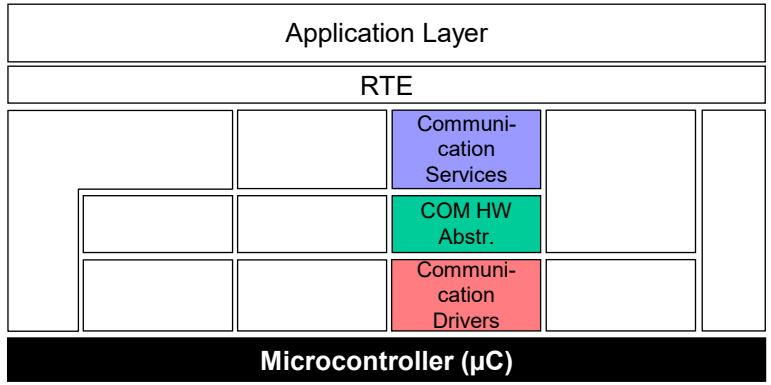
Provide a uniform interface to the LIN network. Hide protocol and message properties from the application.

Properties:

The **LIN Communication Services** contain:

- An ISO 17987 compliant communication stack with
  - Schedule table manager to handle requests to switch to other schedule tables (for LIN master nodes)
  - Communication handling of different LIN frame types
  - Transport protocol, used for diagnostics
  - A WakeUp and Sleep Interface
- An underlying LIN Driver:
  - Implementing LIN protocol and accessing the specific hardware
  - Supporting both simple UART and complex frame based LIN hardware

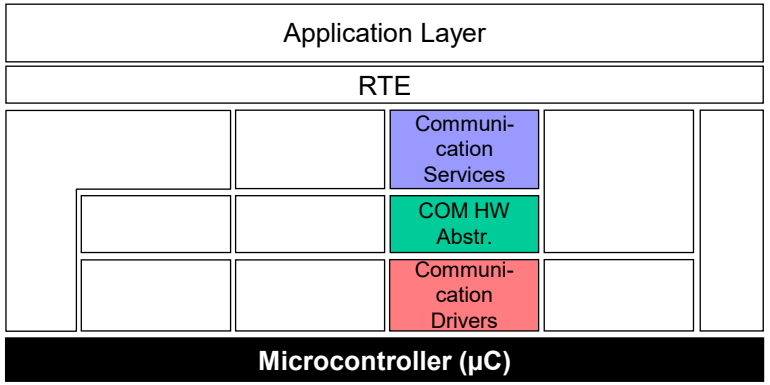
Architecture – Content of Software Layers
Communication Stack – LIN



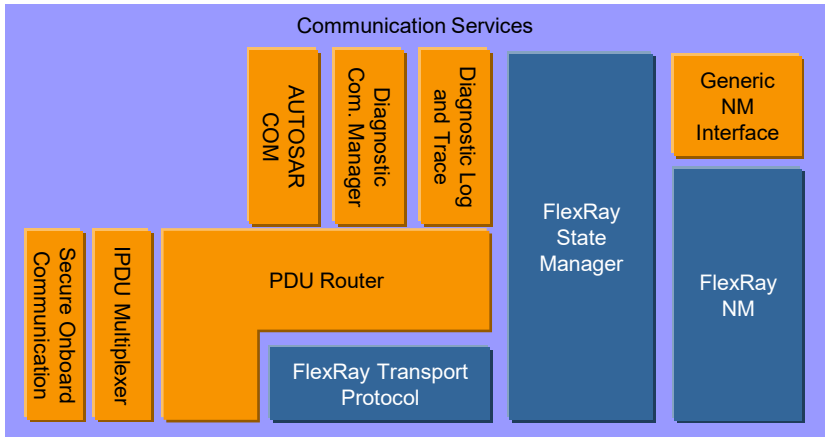
Note: Integration of LIN into AUTOSAR:

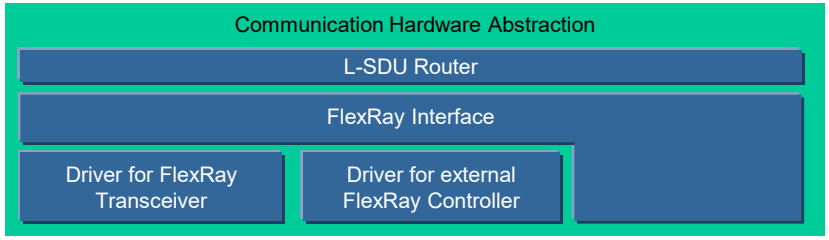
- LIN Interface controls the WakeUp/Sleep API and allows the slaves to keep the bus awake (decentralized approach).
- The communication system specific LIN State Manager handles the communication dependent Start-up and Shutdown features. Furthermore it controls the communication mode requests from the 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).

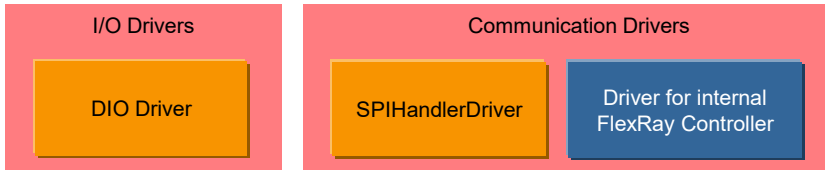
Architecture – Content of Software Layers
Communication Stack – FlexRay

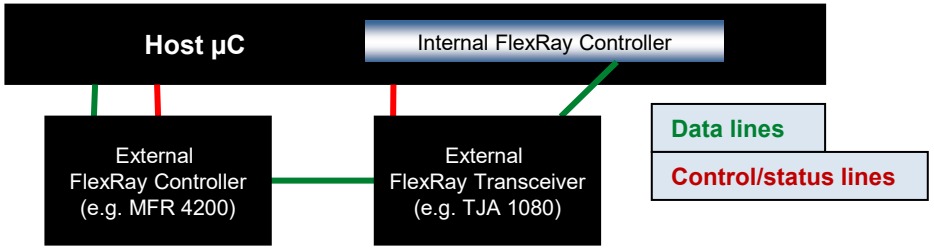


Example:







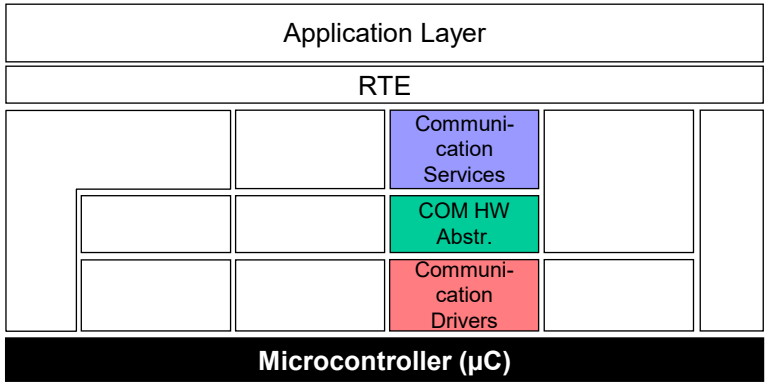


The **FlexRay Communication Services** are a group of modules for vehicle network communication with the communication system FlexRay.

Task:

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

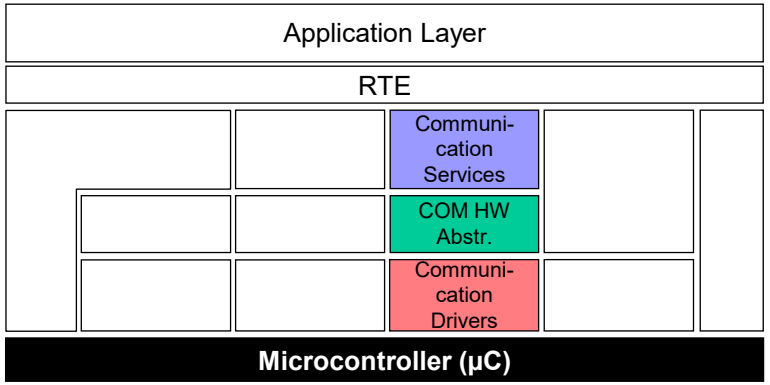
Architecture – Content of Software Layers
Communication Stack – FlexRay



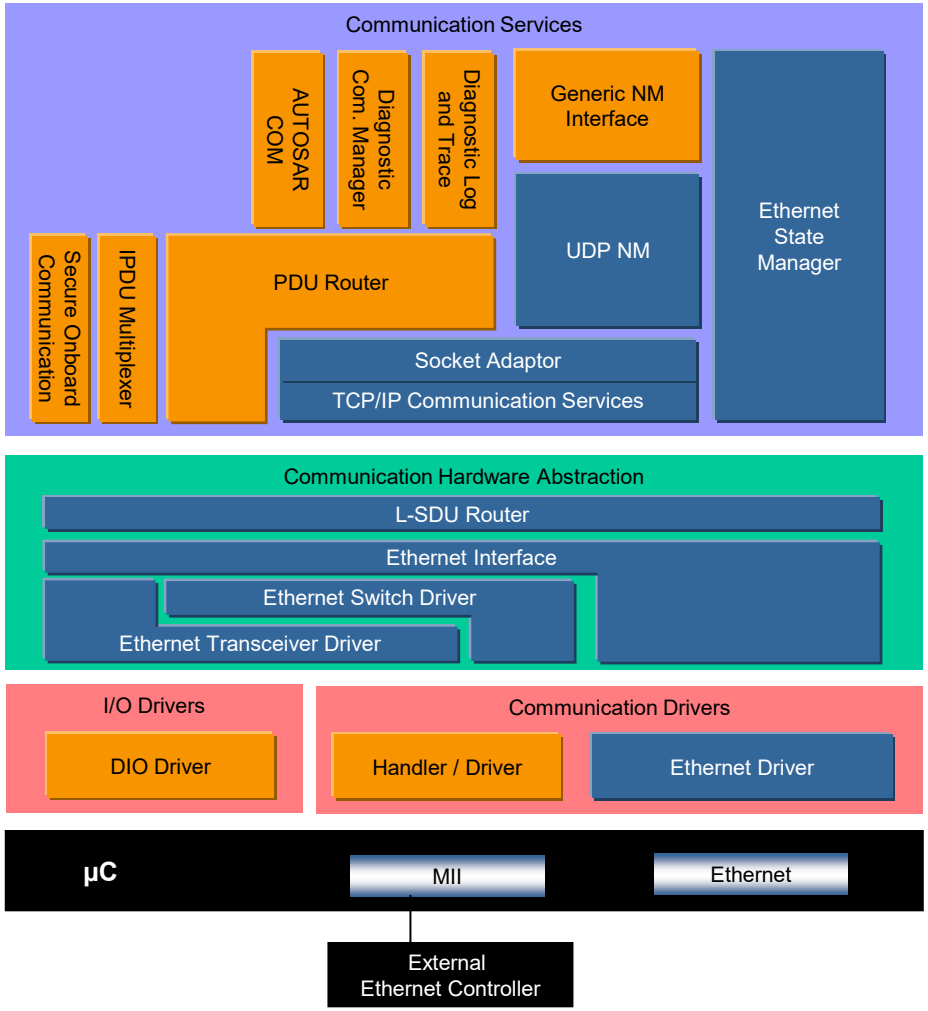
Properties:

- 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.

Architecture – Content of Software Layers
Communication Stack – TCP/IP



Example:

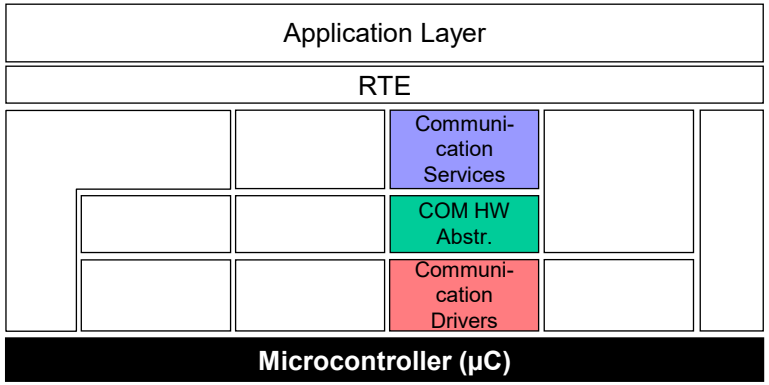


The **TCP/IP Communication Services** are a group of modules for vehicle network communication with the communication system TCP/IP.

Task:

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

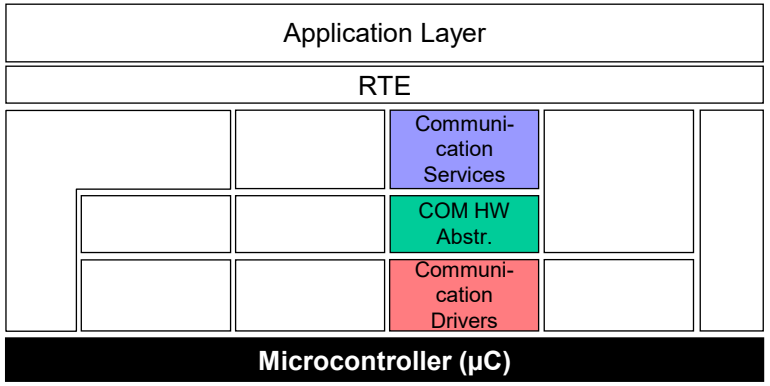
Architecture – Content of Software Layers
Communication Stack – TCP/IP



Properties:

- The TcpIp module implements the main protocols of the TCP/IP protocol family (TCP, UDP, IPv4, IPv6, ARP, ICMP, DHCP) and provides dynamic, socket based communication via Ethernet.
- The Socket Adaptor module (SoAd) is the sole upper layer module of the TcpIp module.

Architecture – Content of Software Layers
Communication Stack – Firewall



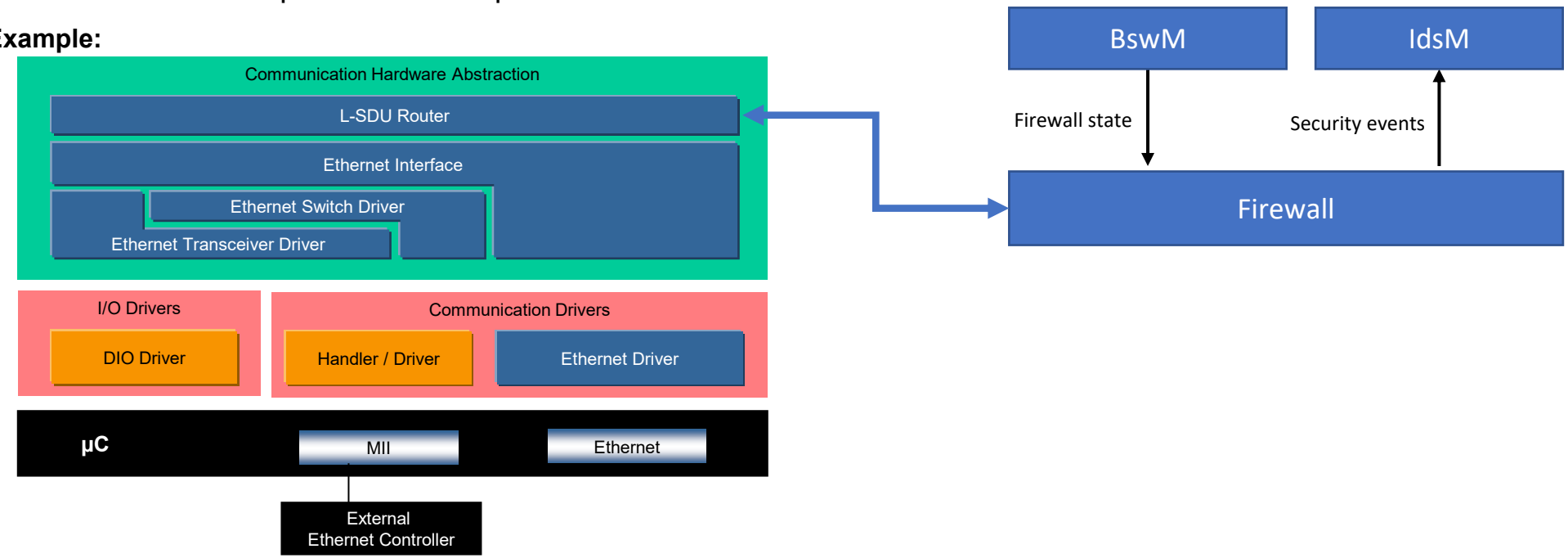
The firewall module protects the AUTOSAR stack from malicious messages by inspecting network packets and filtering them based on a pre-defined ruleset.

The firewall supports network packet inspection on 3 different levels

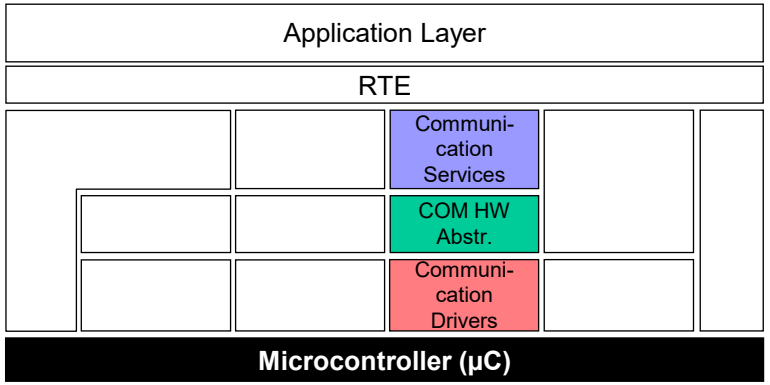
- Stateless packet inspection
- Stateful packet inspection
- Deep packet inspection

The firewall is connected to the IdsM module to raise security events in the case of unexpected network packets

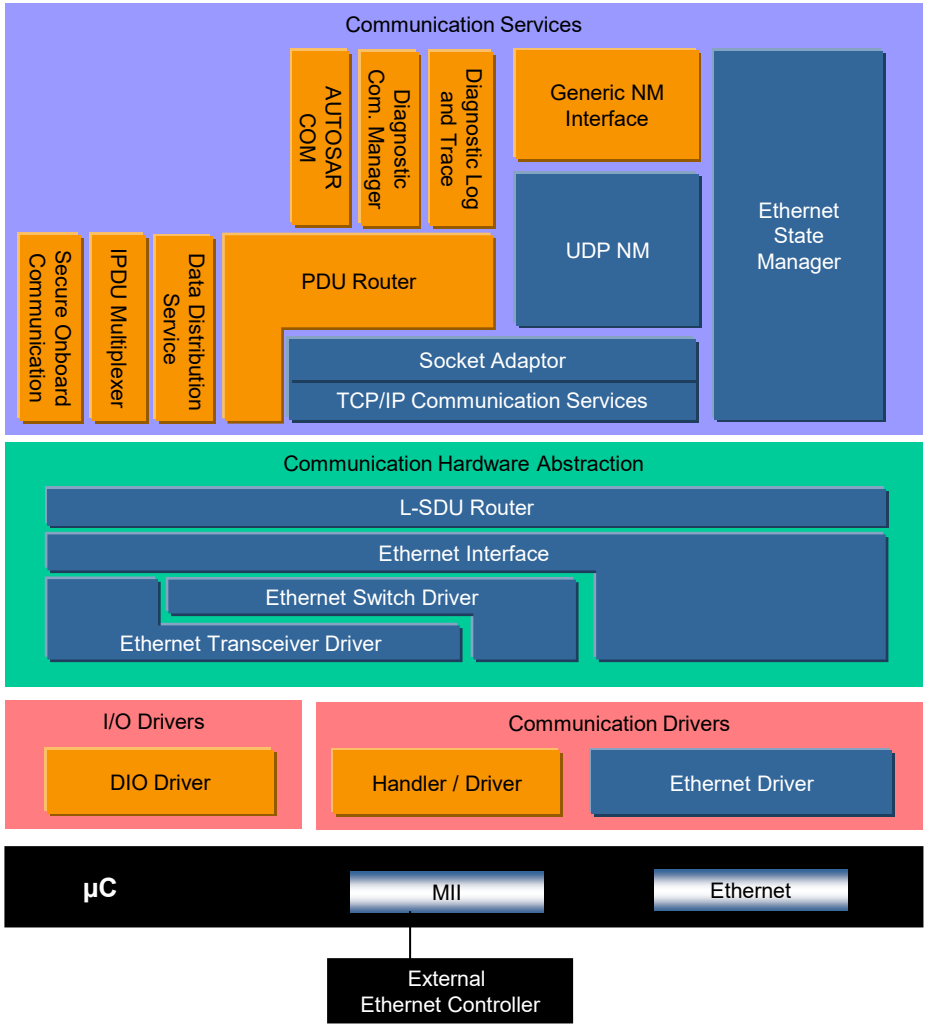
Example:



Architecture – Content of Software Layers
Communication Stack – DDS



Example:



The **Data Distribution Services** is a module for data-oriented and service-oriented vehicle network communication.

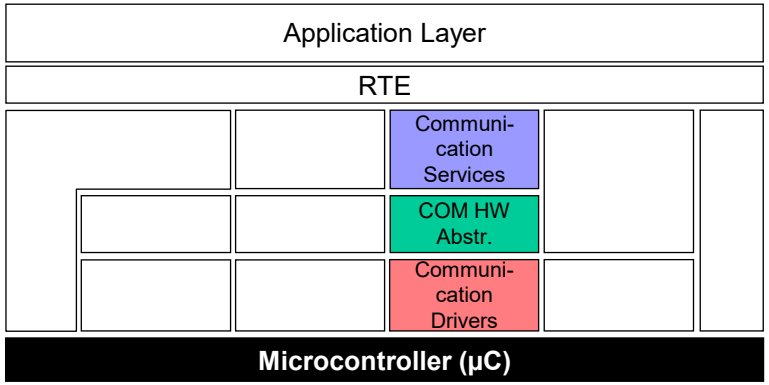
Task:

- Provide the DDS standard interfaces.

The DDS module supports:

- Signal Base Publisher/Subscriber communication path
- QoS handling
- Full static configuration

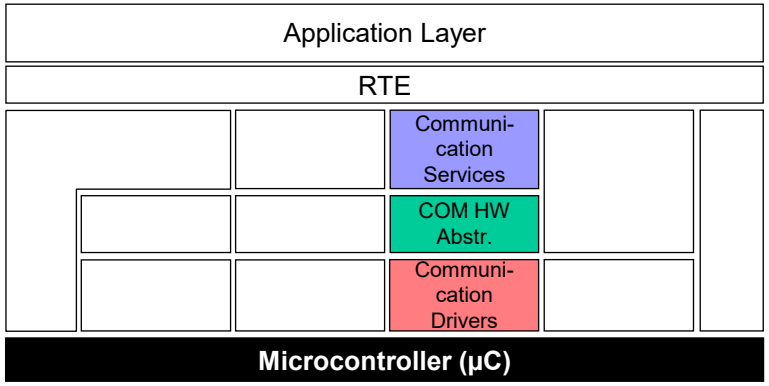
Architecture – Content of Software Layers
Communication Stack – DDS



Properties:

- The DDS module supports the key features of the Object Management Group (OMG) DDS standard, including SPDP and SEDP discovery protocols, the extended SOA relying on DDS-RPC and Service Discovery compliant with AUTOSAR AP.
- The Socket Adaptor module (SoAd) is the sole module able to handle the DDS-PDUs by means of the PDU Router (PduR).
- The DDS module provides E2E features and security services itself.

Architecture – Content of Software Layers
Communication Stack - ChrgM



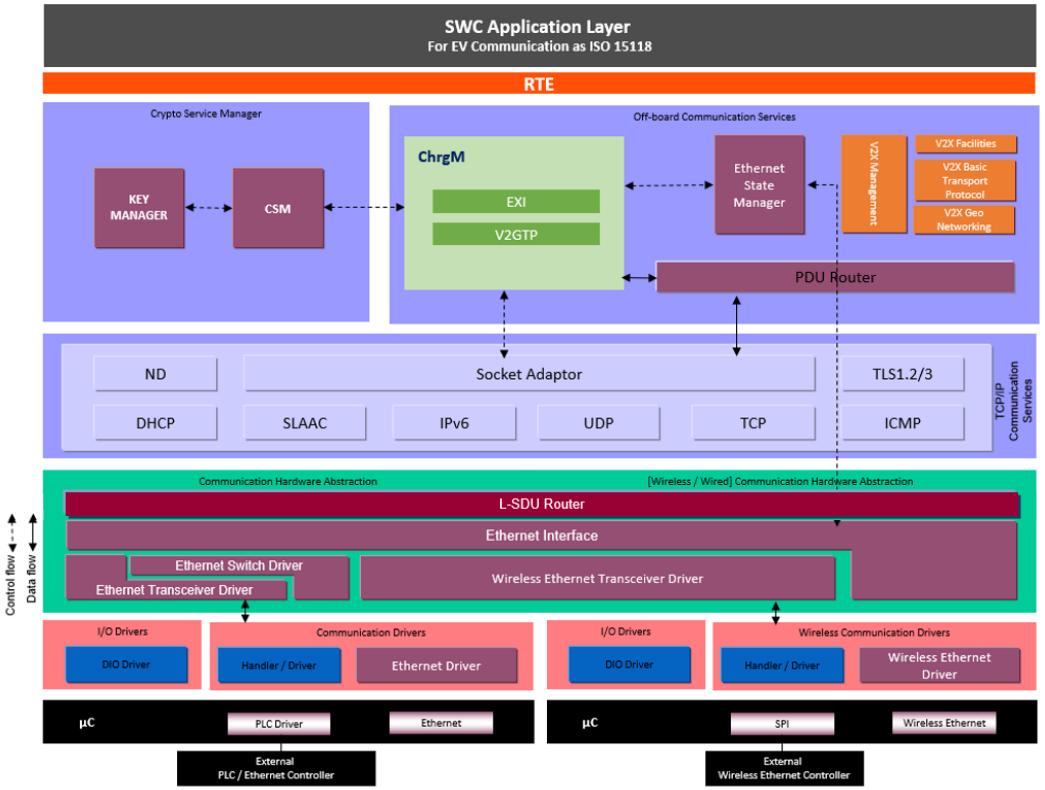
The Charging Manager (ChrgM) belongs to Communication Services of the AUTOSAR Layered Architecture.

Task:

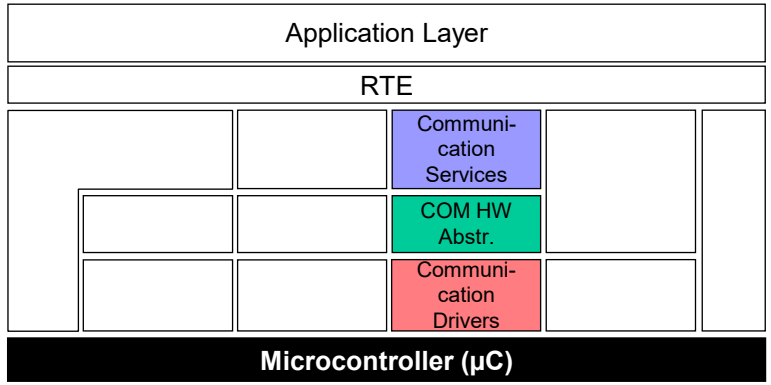
- ChrgM controls the charging process between the EV and the EVSE as per ISO 15118-2.
- ChrgM communicates with different BSW modules such as the PduR, SoAd, Csm, KeyM, BswM, to enable the charging process.
- Provides ports which can be used by the SWCs which implement the application part of charging process.
- Provides V2GTP communication protocol
- Provides EXI encoding & decoding of messages
- ChrgM provides error handling mechanism and timers for managing communication between EV and EVSE.

ChrgM consists of two submodules: V2GTP and EXI.

- V2GTP (vehicle to grid standard protocol): this submodule formats data into a PDU.
- EXI (efficient XML interchange): this submodule converts data into byte streams as per the W3C1.0 recommendation.



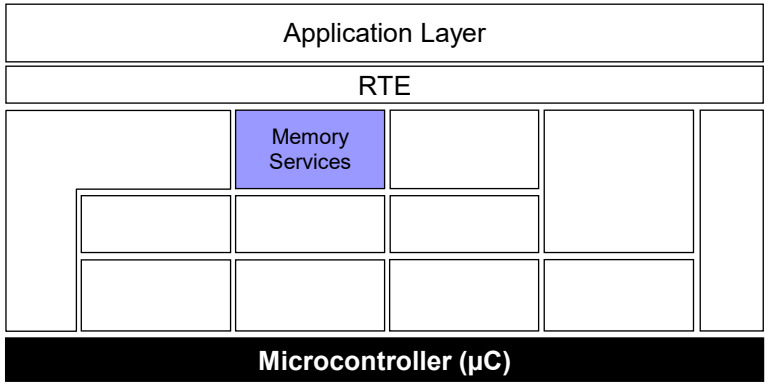
Architecture – Content of Software Layers
Communication Stack – General



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).
- Multi I-PDU to container mapping provides the possibility to combine several I-PDUs into one larger (container-)I-PDU to be transmitted in one (bus specific) frame.
- Upper Interface: µC, ECU hardware and network type independent.
- For refinement of GW architecture please refer to “Example Communication”

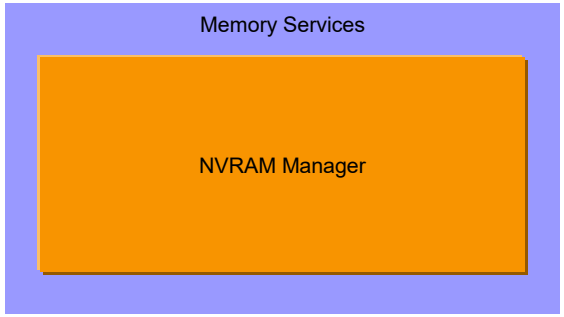
Architecture – Content of Software Layers
Services: Memory Services



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.

Example:

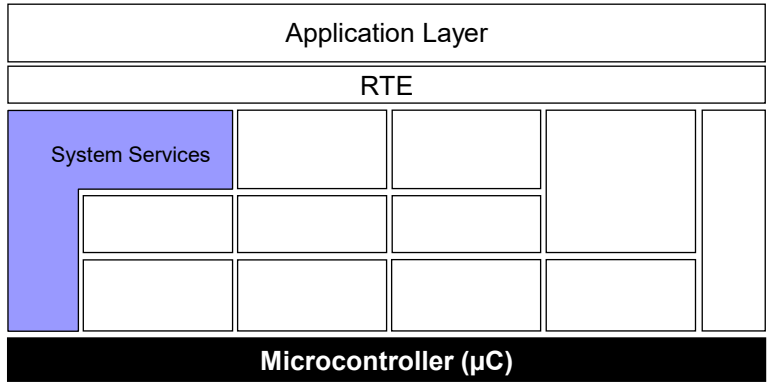


Properties:

Implementation: µC and ECU hardware independent, highly configurable

Upper Interface: µC and ECU hardware independent specified and implemented according to AUTOSAR (AUTOSAR interface)

Architecture – Content of Software Layers
Services: System Services

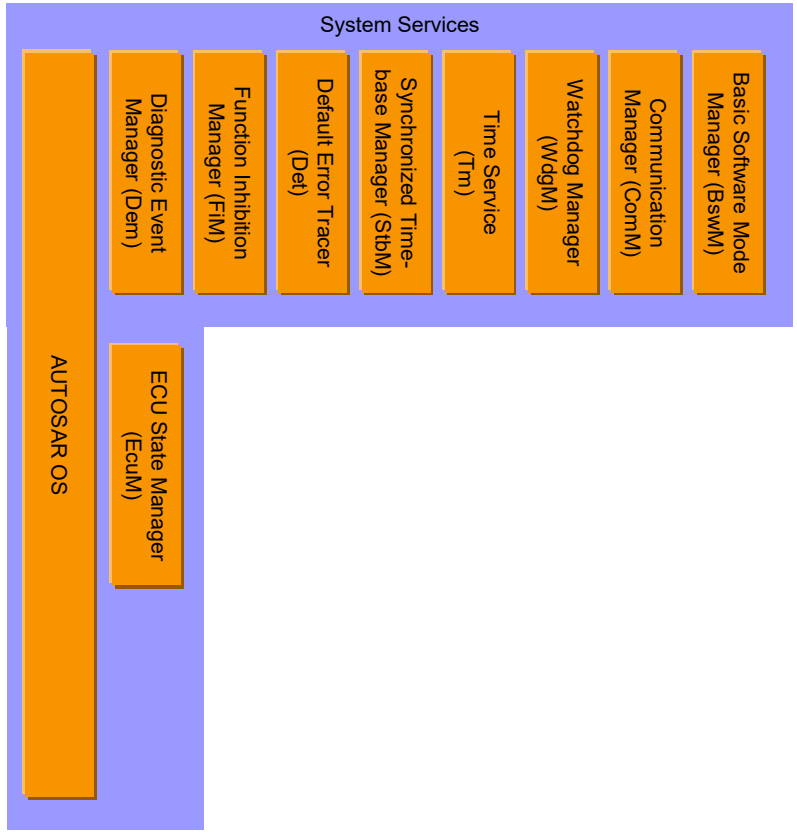


The **System Services** are a group of modules and functions which can be used by modules of all layers. Examples are Real Time Operating System (which includes timer services) and Error Manager.

Some of these services are:

- µC dependent (like OS), and may support special µC capabilities (like Time Service),
- partly ECU hardware and application dependent (like ECU State Manager) or
- hardware and µC independent.

Example:



Task:

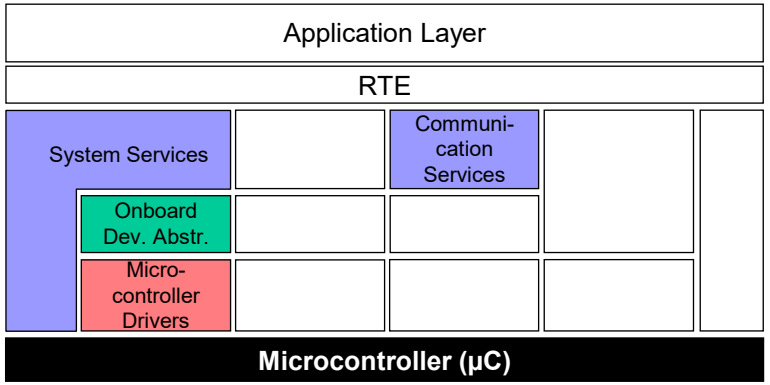
Provide basic services for application and basic software modules.

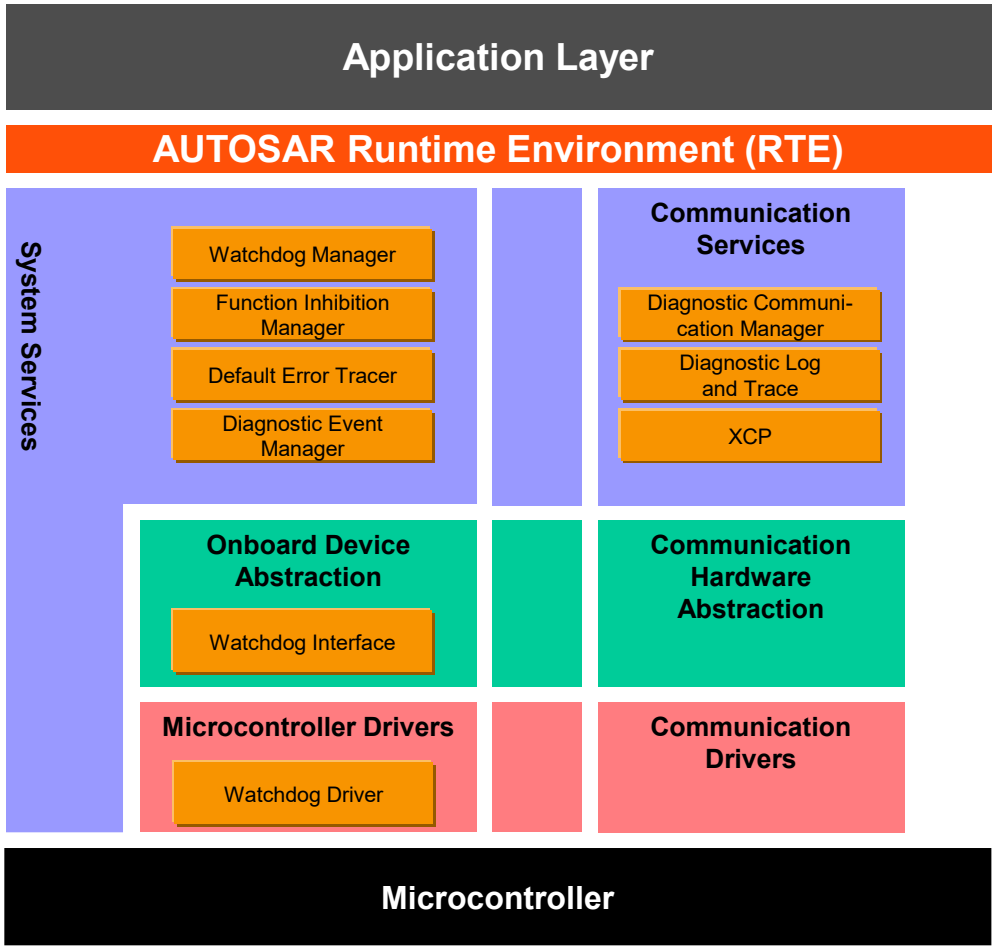
Properties:

Implementation: partly µC, ECU hardware and application specific

Upper Interface: µC and ECU hardware independent

Architecture – Content of Software Layers
Error Handling, Reporting and Diagnostic



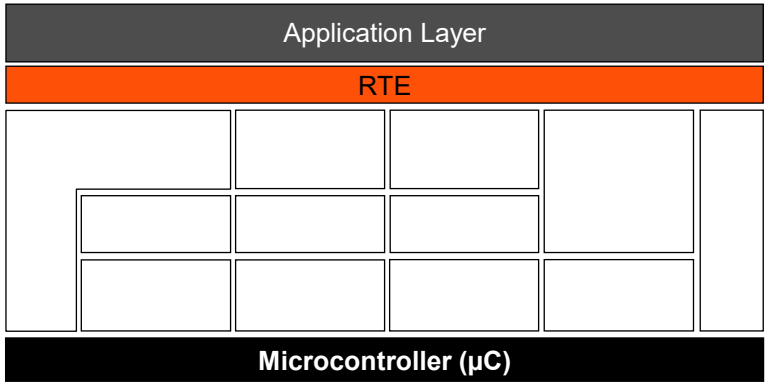


There are dedicated modules for different aspects of error handling in AUTOSAR. E.g.:

- 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.

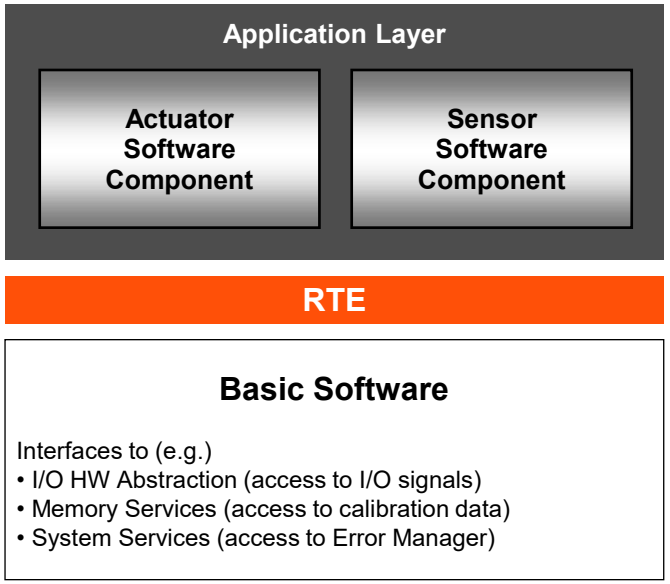
- All detected development errors in the Basic Software are reported to **Default Error Tracer**.
- The **Diagnostic Communication Manager** provides a common API for diagnostic services
- etc.

Architecture – Content of Software Layers
Application Layer: Sensor/Actuator Software Components



The **Sensor/Actuator AUTOSAR Software Component** is a specific type of AUTOSAR Software Component for sensor evaluation and actuator control. Though not belonging to the AUTOSAR Basic Software, it is described here due to its strong relationship to local signals. It has been decided to locate the Sensor/Actuator SW Components above the RTE for integration reasons (standardized interface implementation and interface description). Because of their strong interaction with raw local signals, relocatability is restricted.

Example:



Task:

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

Properties:

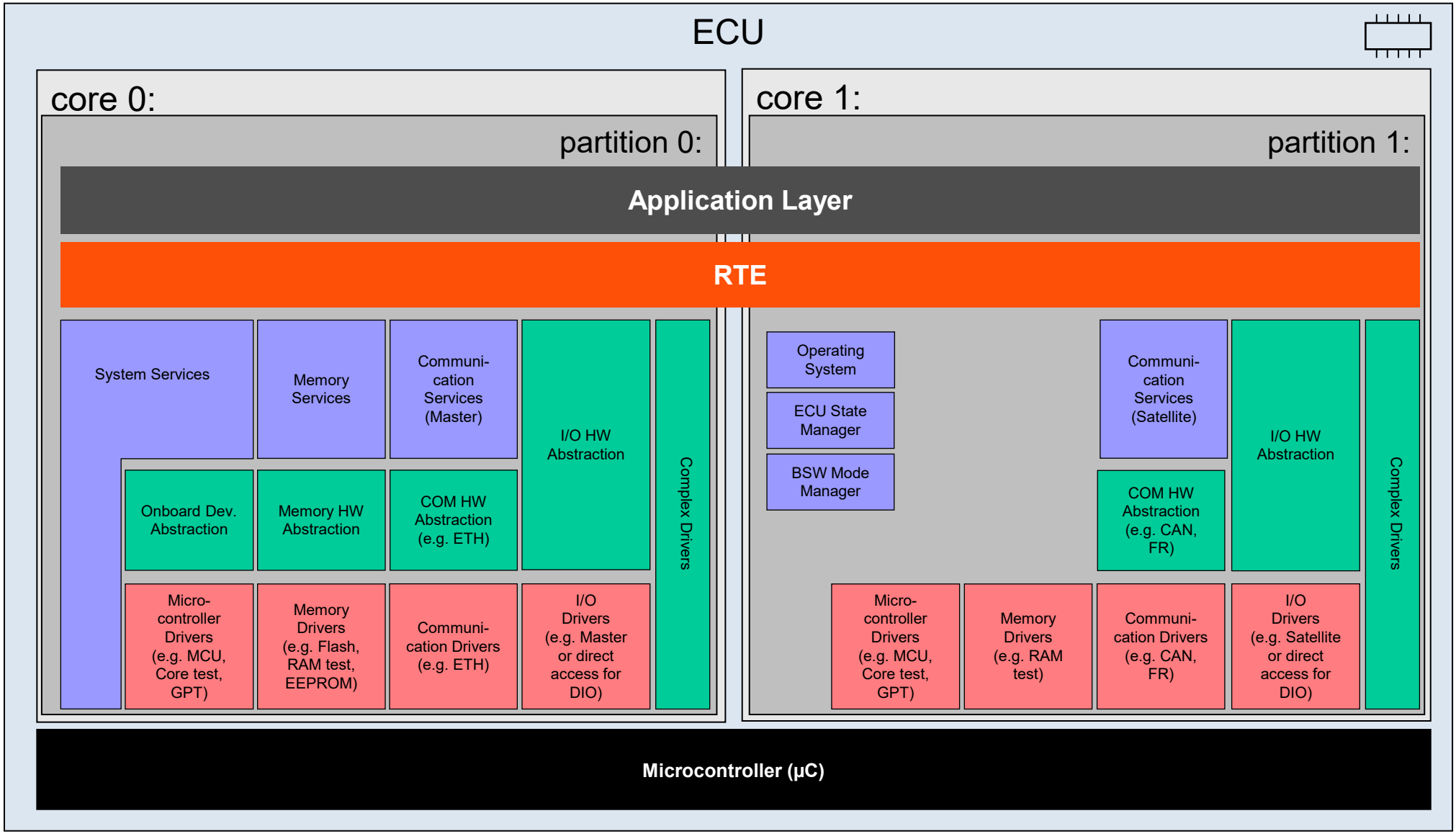
Implementation: µC and ECU HW independent, sensor and actuator dependent

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. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. Interfaces: General Rules
    7. Interfaces: Interaction of Layers
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Architecture – Content of Software Layers
Example of a Layered Software Architecture for Multi-Core Microcontroller

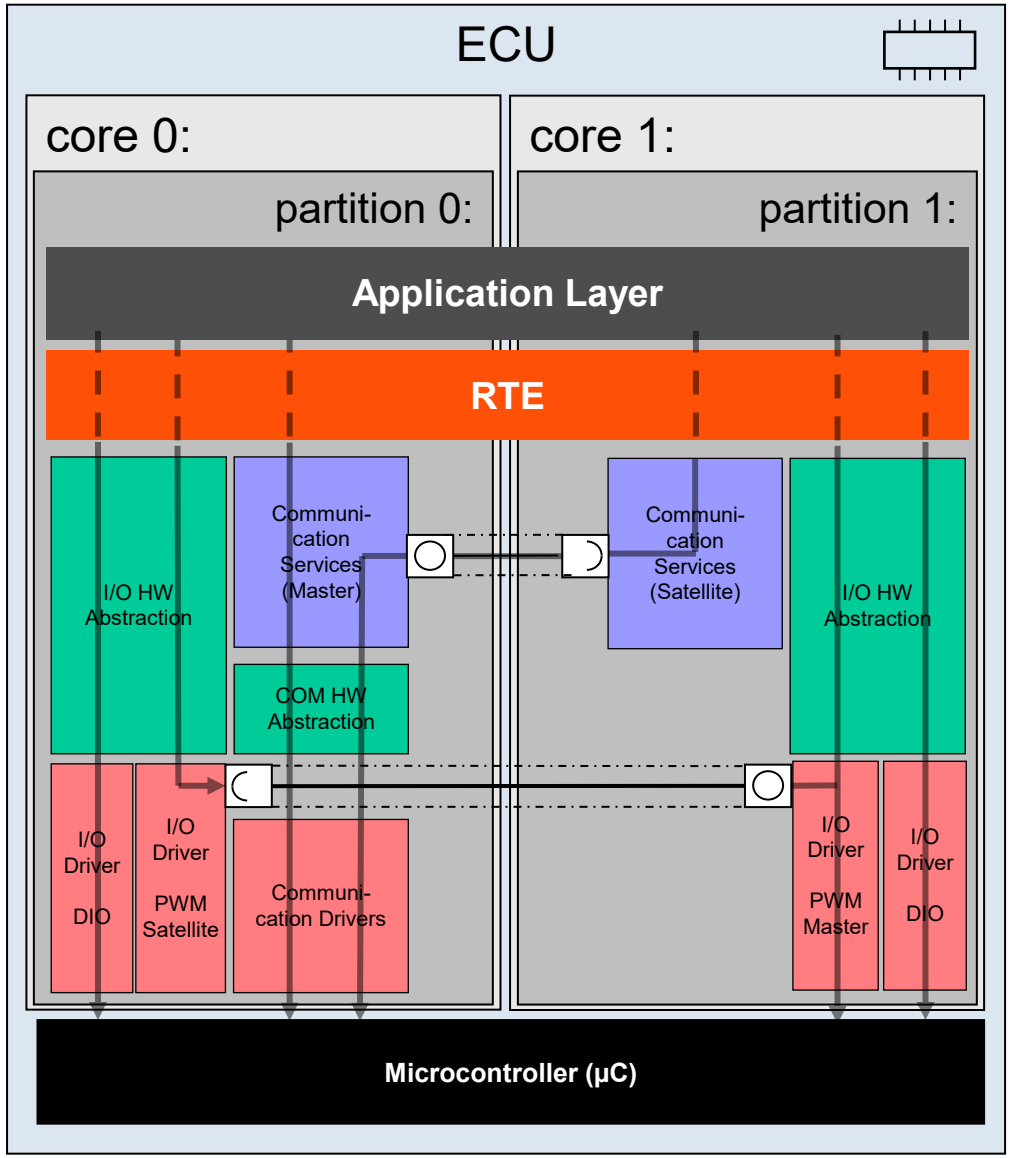
Example: an ECU with a two core microcontroller



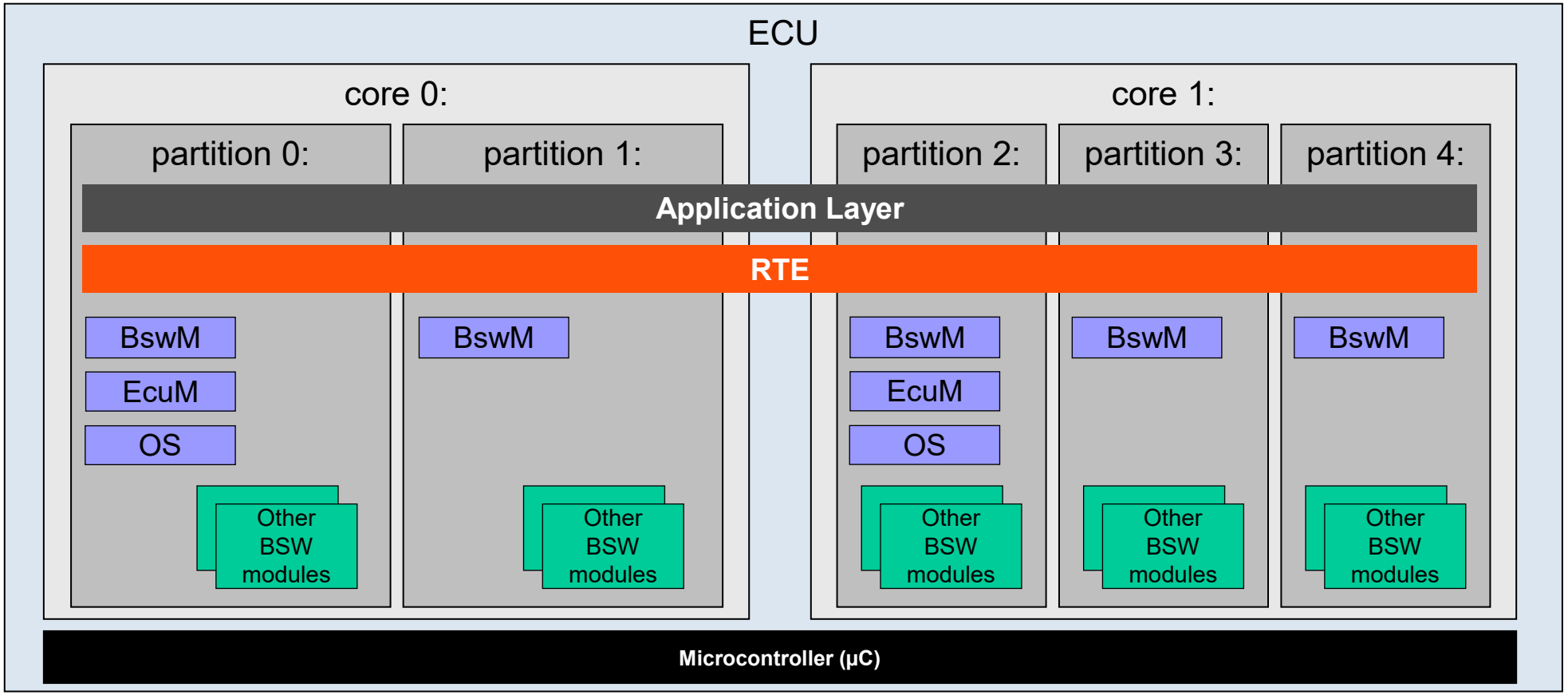
Architecture – Content of Software Layers
Detailed View of Distributed BSW Modules

Example: an ECU with a two core microcontroller

- BSW modules can be distributed across several partitions and cores. All partitions share the same code.
- Modules can either be completely identical on each partition, as shown for the DIO driver out of I/O stack in the figure.
- As an alternative, they can use core-dependent branching to realize different behavior. Com service and PWM driver use master-satellite communication for processing a call to the master from the according satellites.
  - The communication between master and satellite is not standardized. For example, it can be based on functions provided by the BSW scheduler or on shared memory.
- The arrows indicate which components are involved in the handling of a service call, depending on the approach to distribution and on the origin of the call.

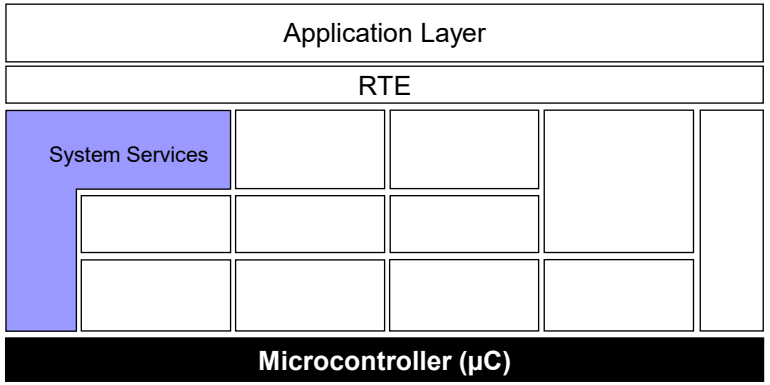


Architecture – Content of Software Layers
Overview of BSW Modules, OS, BswM and EcuM on Multiple Partitions



- Basic Software Mode Manager (BswM) in every partition that runs BSW modules
  - all these partitions are trusted
- One EcuM per core (each in a trusted partition)
- EcuM on that core that gets started via the boot-loader is the master EcuM
  - Master EcuM starts all Satellite EcuMs

Architecture – Content of Software Layers
Scope: Multi-Core System Services



- 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.
- BSW modules can be executable on several cores, such as the ComM in the figure. The core responsible for executing a service is determined at runtime.
- Every core runs a kind of ECU state management.

Example: an ECU with a two core microcontroller

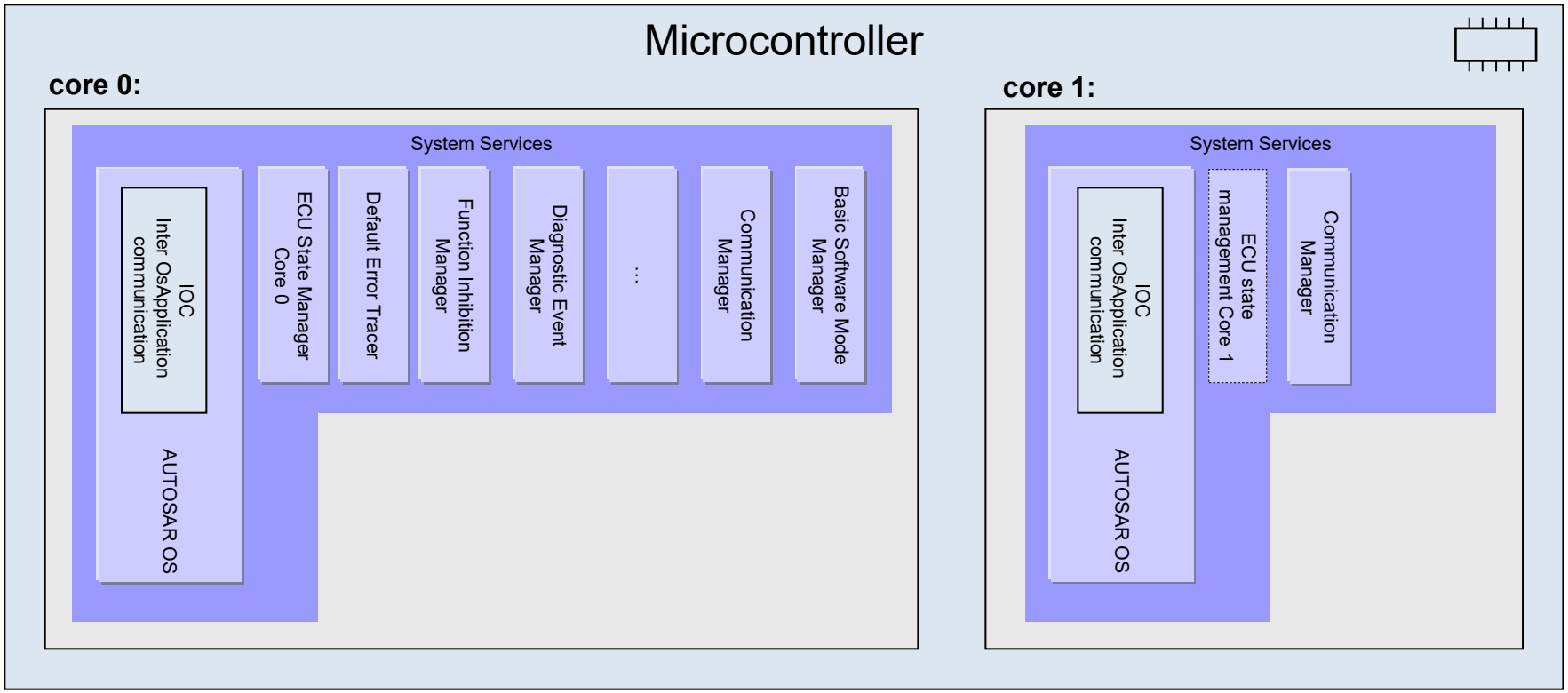
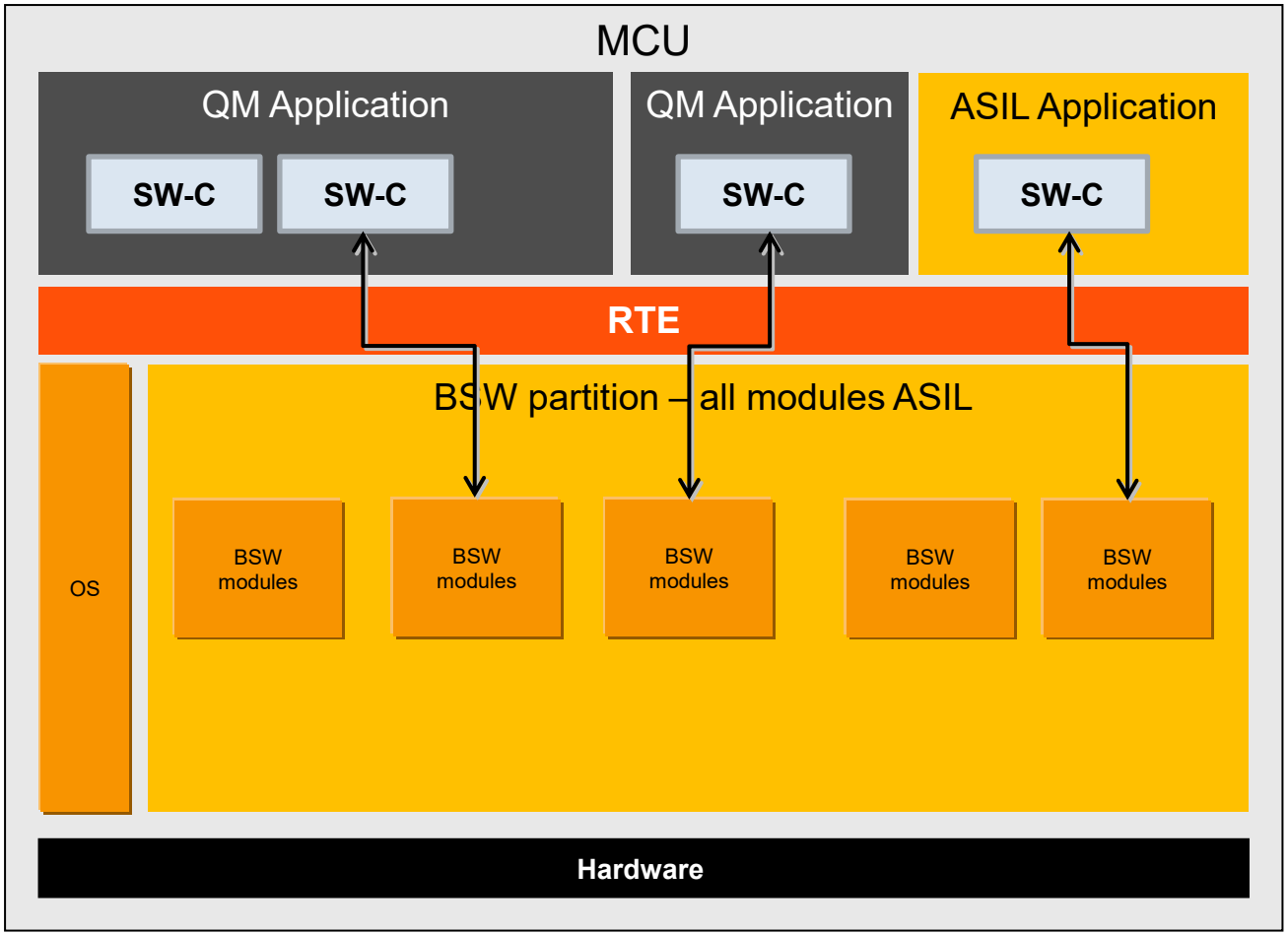


Table of contents

1. **Architecture**
    1. Overview of Software Layers
    2. Content of Software Layers
    3. Content of Software Layers in Multi-Core Systems
    4. **Content of Software Layers in Mixed-Critical Systems**
    5. Overview of Modules
    6. Interfaces: General Rules
    7. Interfaces: Interaction of Layers
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Architecture – Content of Software Layers
Overview of AUTOSAR safety handling

- AUTOSAR offers a flexible approach to support safety relevant ECUs. Two methods can be used:
  1. All BSW modules are developed according to the required ASIL
  2. Selected modules are developed according to ASIL. ASIL and non-ASIL modules are separated into different partitions (BSW distribution)



Example for usage of method (1)

Note: The partitions are based on OS-Applications. The TRUSTED attribute of the OS-Application is not related to ASIL/non-ASIL.

Architecture – Content of Software Layers
AUTOSAR BSW distribution for safety systems

- Example of using different BSW partitions
  - Watchdog stack is placed in a own partition
  - ASIL and non-ASIL SW-Cs can access WdgM via RTE
  - Rest of BSW is placed in own partition

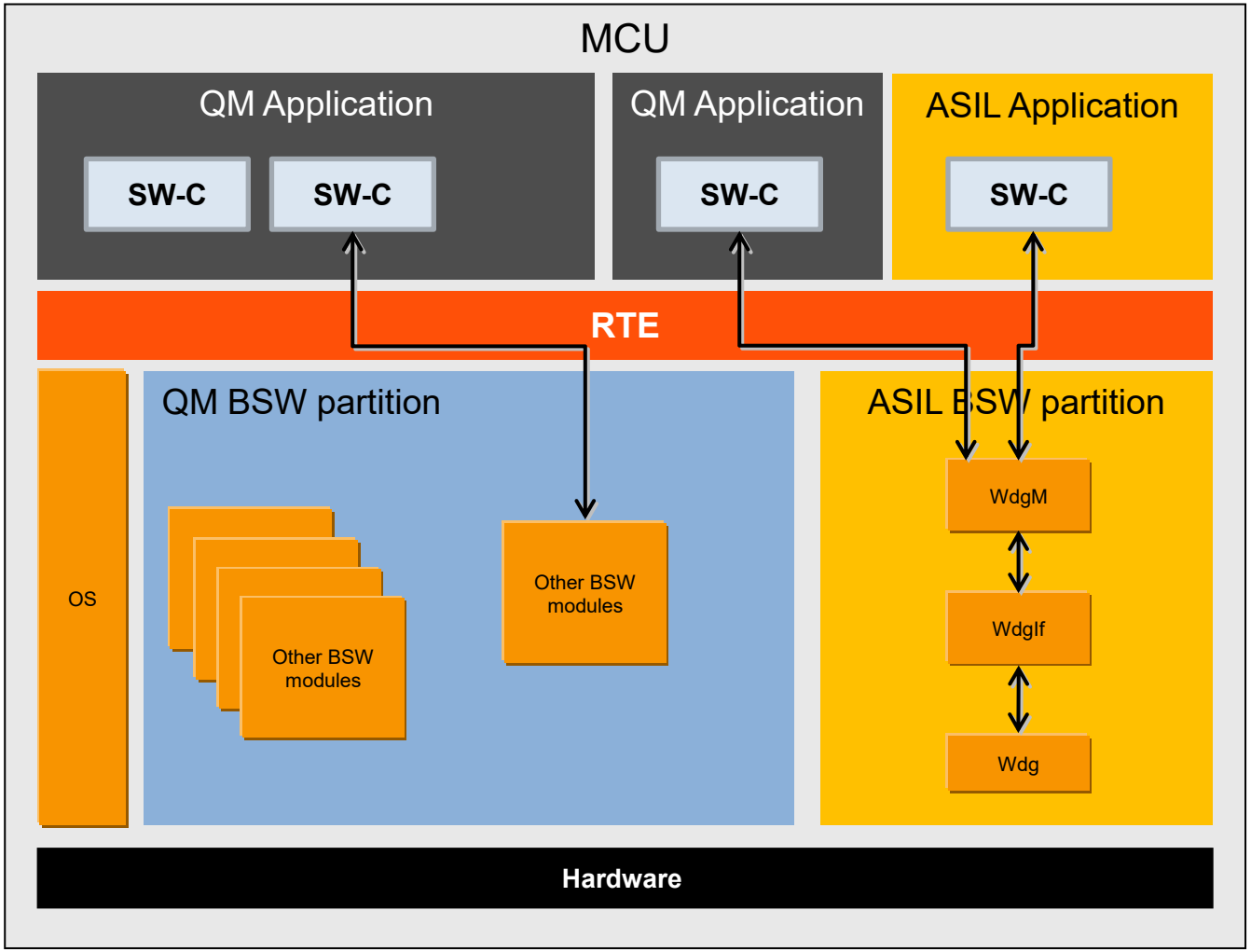
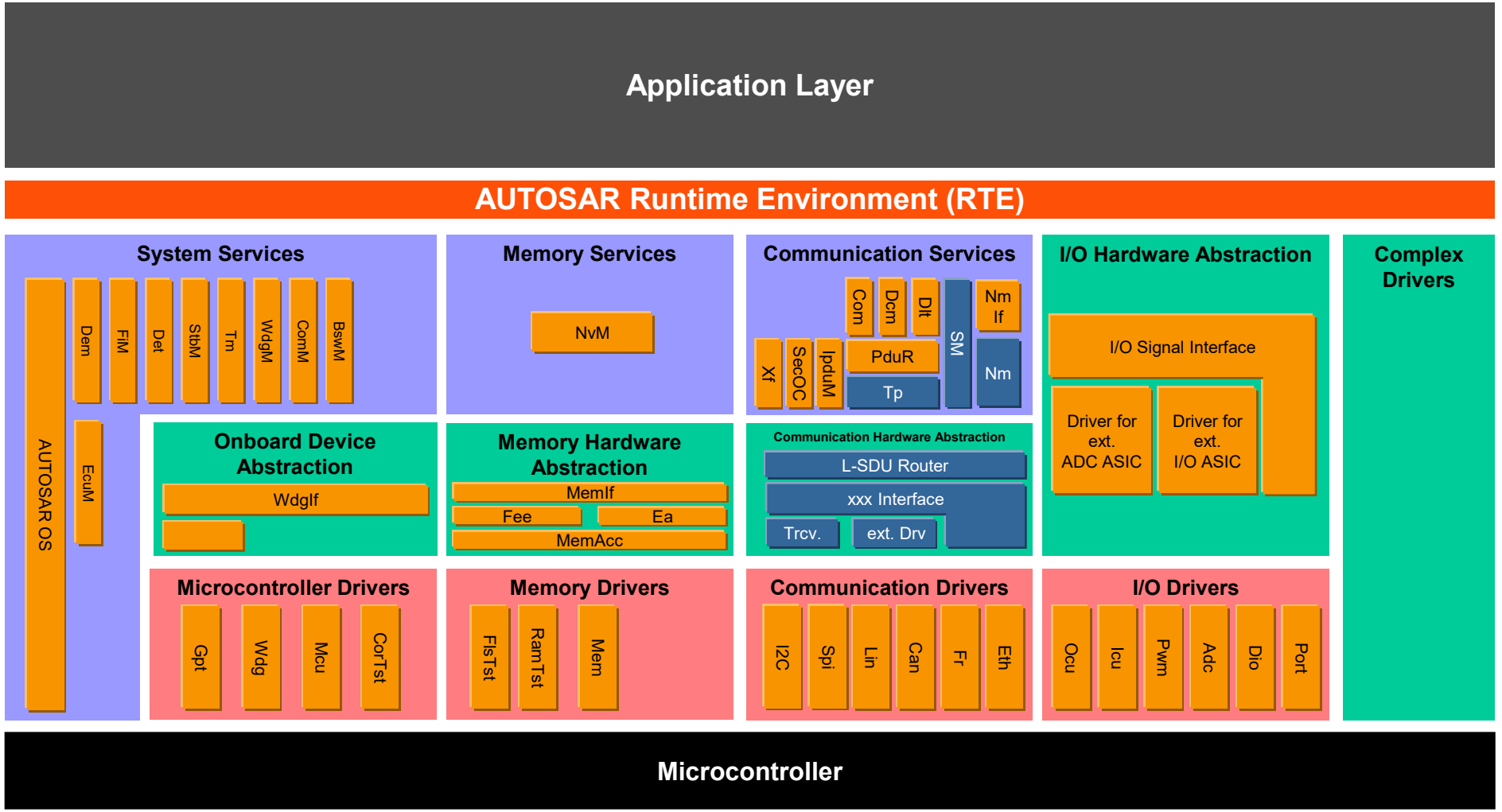


Table of contents

1. **Architecture**
    1. Overview of Software Layers
    2. Content of Software Layers
    3. Content of Software Layers in Multi-Core Systems
    4. Content of Software Layers in Mixed-Critical Systems
    5. **Overview of Modules**
    6. Interfaces: General Rules
    7. Interfaces: Interaction of Layers
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Architecture
Overview of Modules – Implementation Conformance Class 3 - ICC3

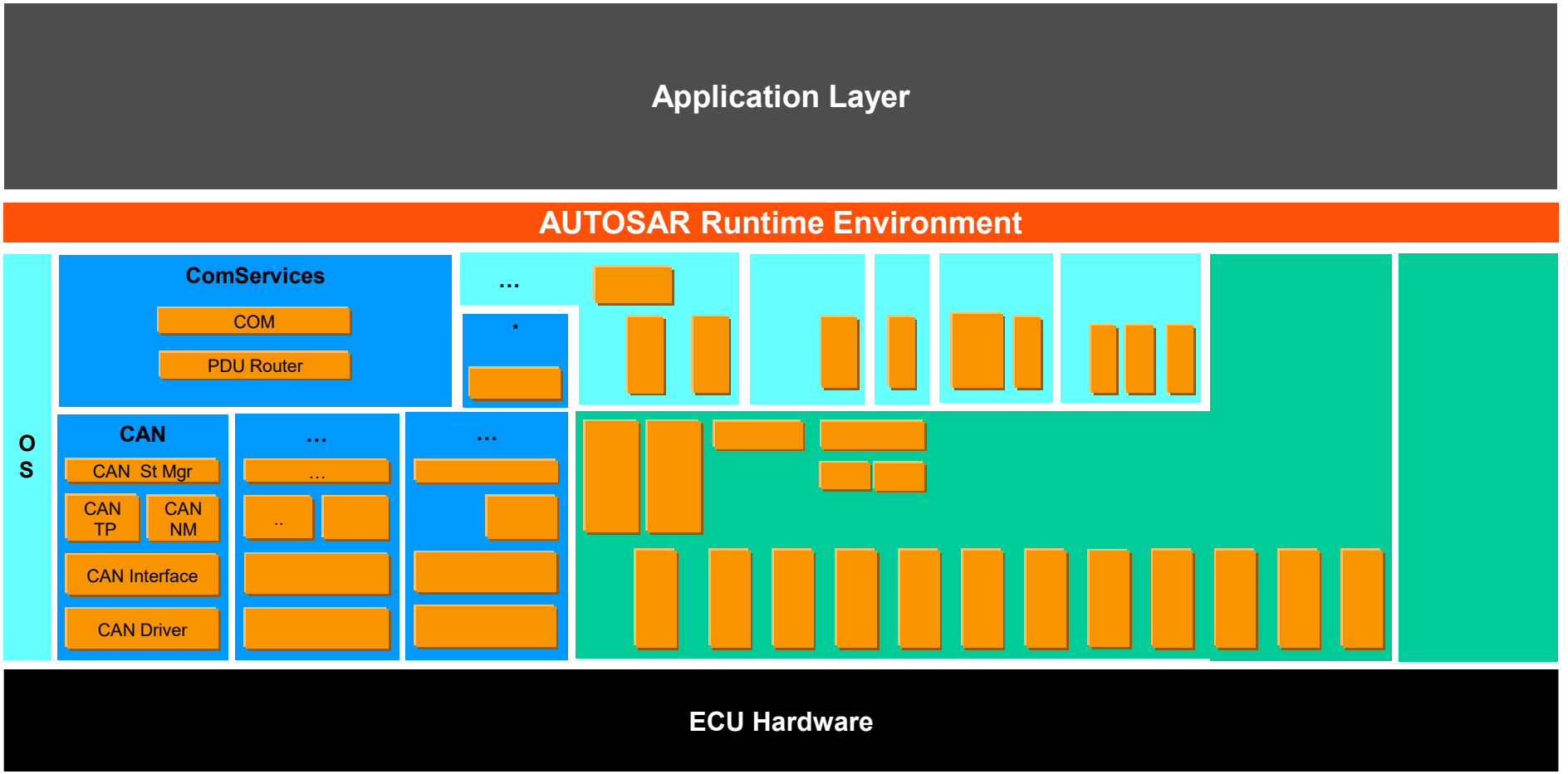
This figure shows the mapping of basic software modules to AUTOSAR layers



Not all modules are shown here

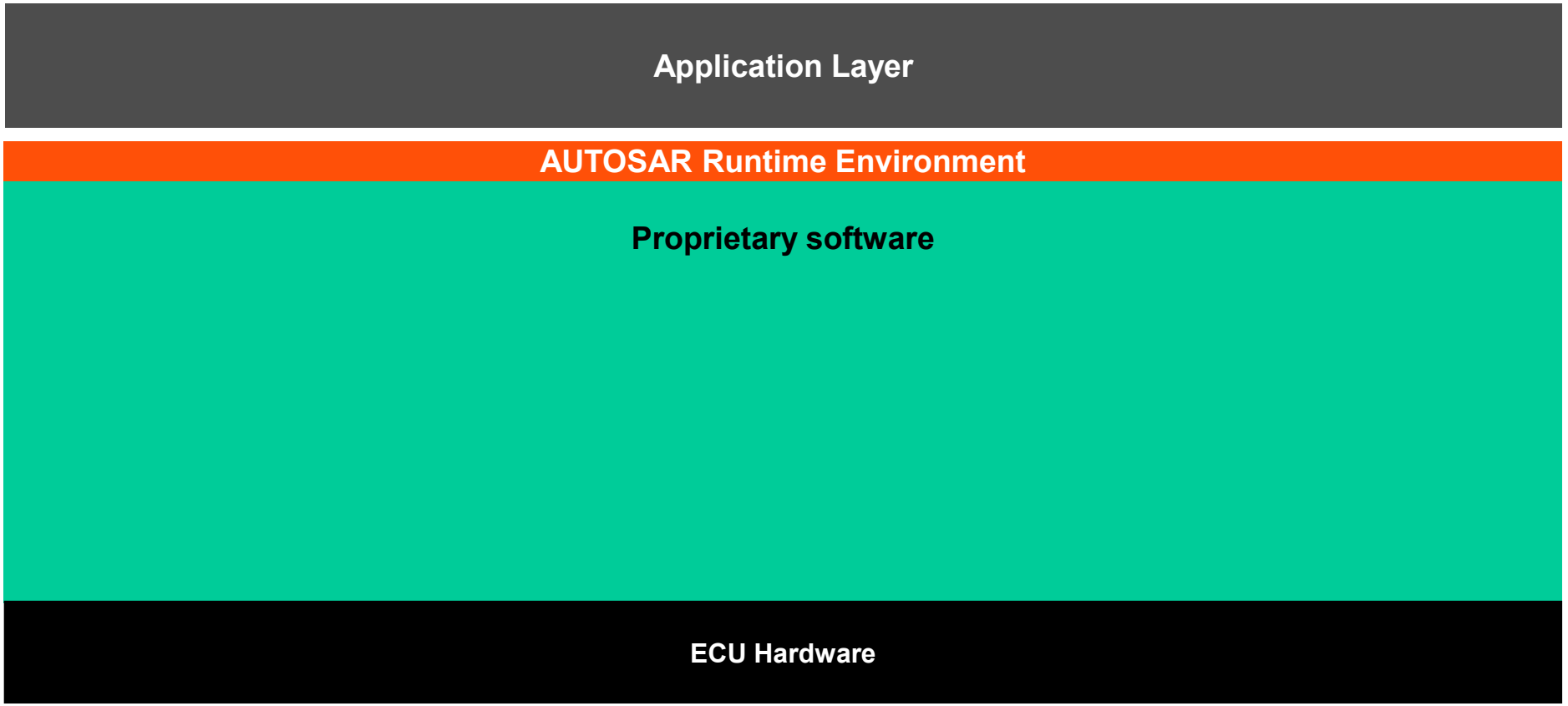
Architecture
Overview of Modules – Implementation Conformance Classes – ICC2

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.



Overview of Modules – Implementation Conformance Classes – ICC1

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.



Architecture
Overview of Modules – Implementation Conformance Classes – behavior to the outside

Basic software (including the RTE) which is AUTOSAR compliant (ICC1-3) has to behave to the outside as specified by the ICC3 module specification.

For example the behavior towards:

- buses,
- boot loaders and
- Applications

Additionally, the ICC1/2 configuration shall be compatible regarding the system description as in ICC3.

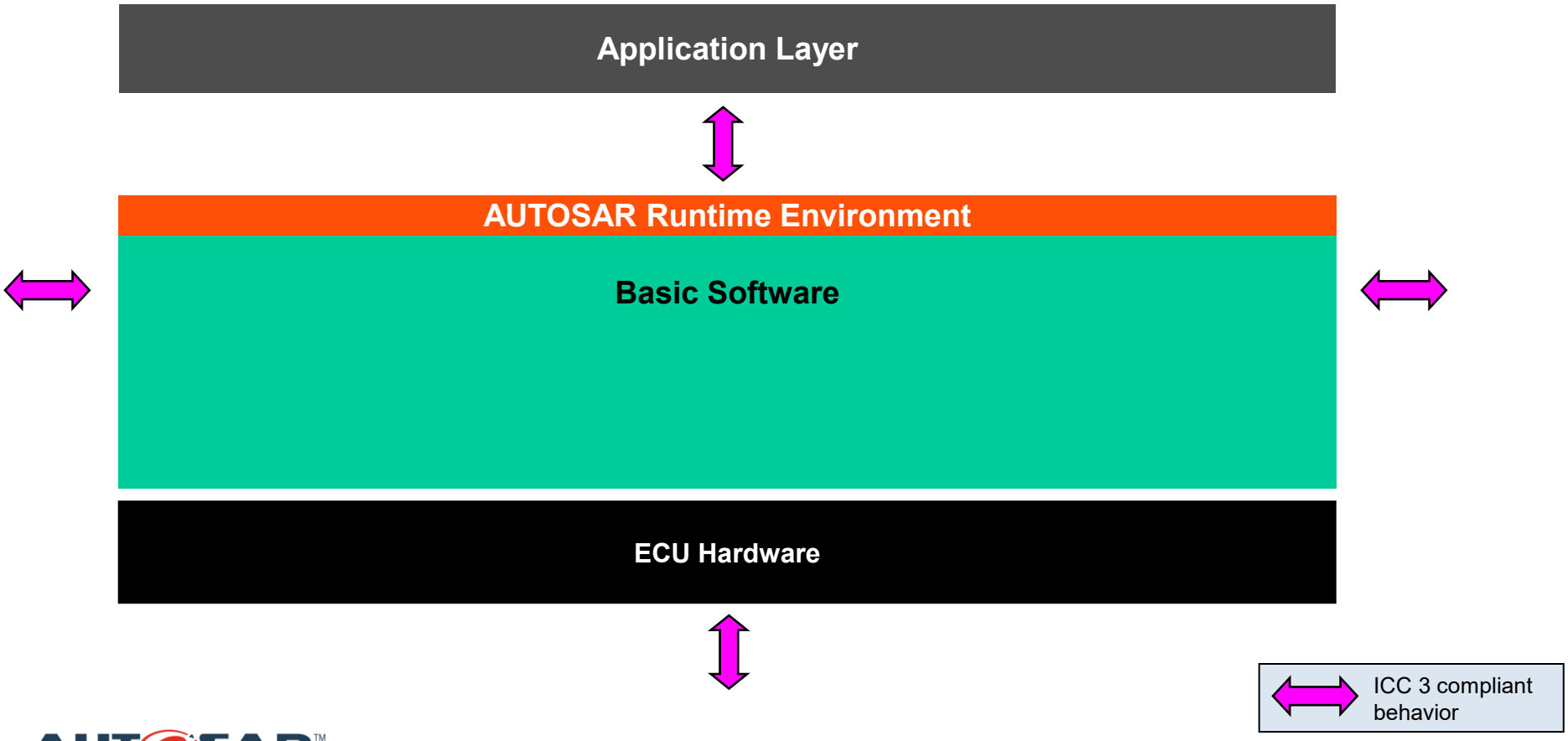


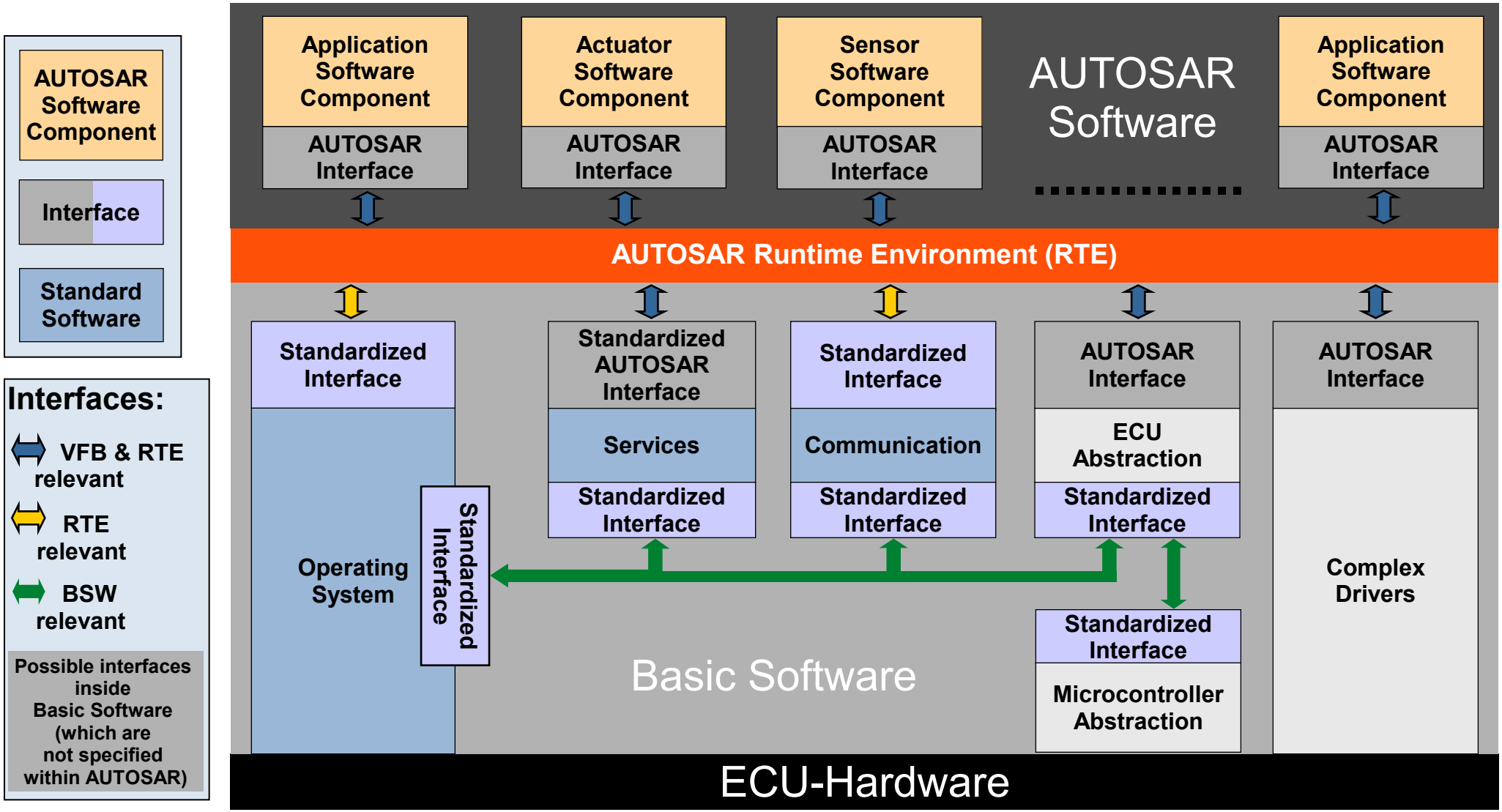
Table of contents

1. **Architecture**
    1. Overview of Software Layers
    2. Content of Software Layers
    3. Content of Software Layers in Multi-Core Systems
    4. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. **Interfaces: General Rules**
    7. Interfaces: Interaction of Layers
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Interfaces
Type of Interfaces in AUTOSAR

AUTOSAR Interface	An "AUTOSAR Interface" 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.
Standardized AUTOSAR Interface	A "Standardized AUTOSAR Interface" is an "AUTOSAR Interface" whose syntax and semantics are standardized in AUTOSAR. The "Standardized AUTOSAR Interfaces" are typically used to define AUTOSAR Services, which are standardized services provided by the AUTOSAR Basic Software to the application Software-Components.
Standardized Interface	A "Standardized Interface" is an API which is standardized within AUTOSAR without using the "AUTOSAR Interface" technique. These "Standardized Interfaces" are typically defined for a specific programming language (like "C"). Because of this, "standardized interfaces" are typically used between software-modules which are always on the same ECU. When software modules communicate through a "standardized interface", it is NOT possible any more to route the communication between the software-modules through a network.

Interfaces
Components and interfaces view (simplified)



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

Interfaces: General Rules
General Interfacing Rules

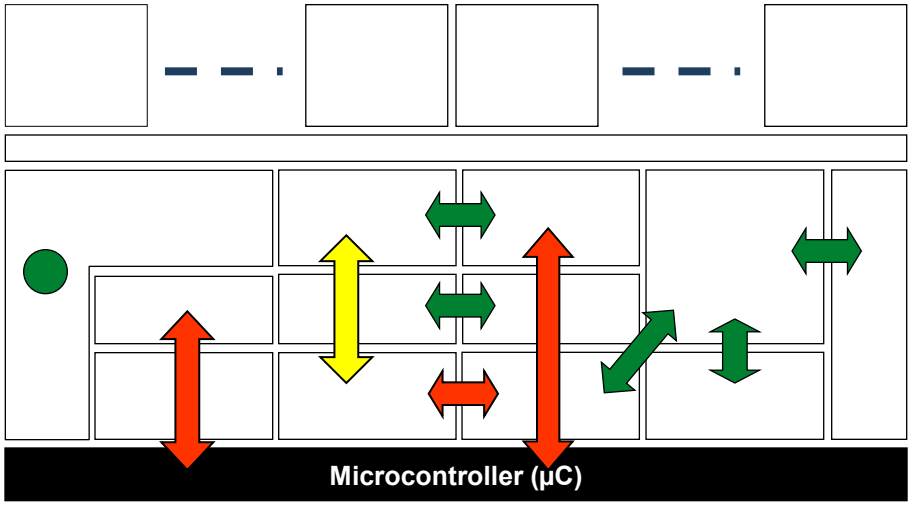
Horizontal Interfaces

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

ECU Abstraction Layer: horizontal interfaces are allowed

A complex driver may use selected other BSW modules

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



Vertical Interfaces

One Layer may access all interfaces of the SW layer below

Bypassing of one software layer should be avoided

Bypassing of two or more software layers is not allowed

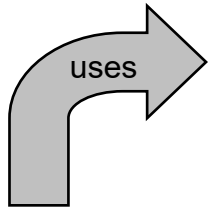
Bypassing of the µC Abstraction Layer is not allowed

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

All layers may interact with system services.

Interfaces: General Rules
Layer Interaction Matrix

This normative matrix shows the allowed interactions between AUTOSAR Basic Software layers



	System Services / OS	Memory Services	Crypto Services	Communication Services	Complex Drivers	I/O Hardware Abstraction	Onboard Device Abstr.	Memory HW Abstraction	Crypto HW Abstraction	Comm. HW Abstraction*	Microcontroller Drivers	Memory Drivers	Crypto Drivers	Communication Drivers*	I/O Drivers	Microcontroller Hardware
SW Components / RTE	✓	✓	✓	✓	✓	✓	×	×	×	×	×	×	×	×	×	×
System Services / OS	✓	✓	✓	✓	Δ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Memory Services	✓	✓	✓	×	Δ	×	×	✓	×	×	×	×	×	×	×	×
Crypto Services	✓	✓	✓	×	Δ	×	×	×	✓	×	×	×	×	×	×	×
Communication Services	✓	✓	✓	✓	Δ	×	×	×	×	✓	×	×	×	×	×	×
Complex Drivers	restricted access -> see the following two slides															
I/O Hardware Abstraction	✓	×	✓	×	×	✓	✓	×	×	✓	✓	×	×	✓	✓	×
Onboard Device Abstr.	✓	×	×	×	×	×	✓	×	×	✓	✓	×	×	✓	✓	×
Memory HW Abstraction	✓	✓	×	×	×	×	✓	✓	×	✓	×	✓	×	✓	×	×
Crypto HW Abstraction	✓	✓	✓	×	×	×	✓	×	✓	×	×	×	×	×	✓	×
Comm. HW Abstraction*	✓	×	×	✓	×	×	✓	×	×	✓	×	×	×	✓	✓	×
Microcontroller Drivers	✓	×	×	×	×	Δ	Δ	×	Δ	×	Δ	×	×	×	Δ	✓
Memory Drivers	✓	×	×	×	×	×	×	Δ	×	×	×	×	×	×	×	✓
Crypto Drivers	✓	✓	×	×	×	×	×	×	Δ	×	×	×	×	×	×	✓
Communication Drivers*	✓	×	×	×	×	×	Δ	×	×	Δ	×	×	×	×	✓	✓
I/O Drivers	✓	×	×	×	×	Δ	Δ	×	Δ	×	Δ	×	×	×	Δ	✓

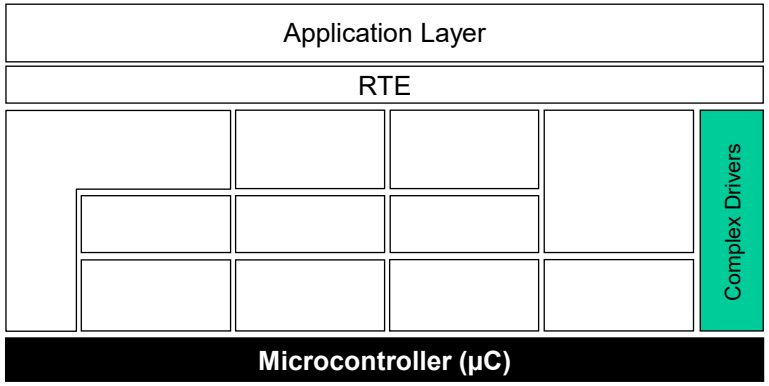
✓ allowed to use
× not allowed to use
Δ restricted use (callback only)

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)

*: includes wired communication

Interfaces
Interfacing with Complex Drivers (1)



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.

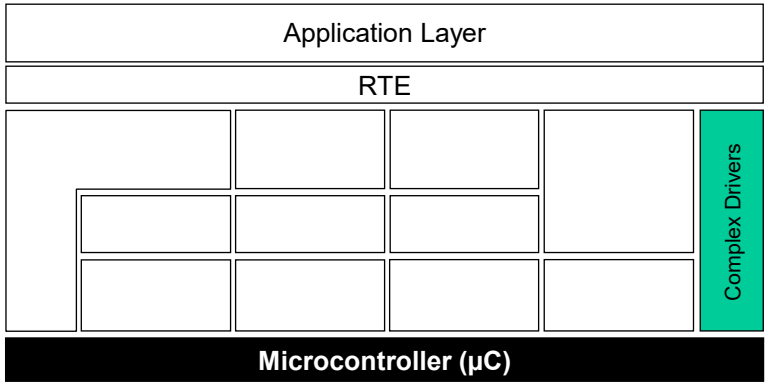
A typical example is the PDU Router: a Complex Driver may implement the interface module of a new bus system. This is already taken care of within the configuration of the PDU Router.

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)

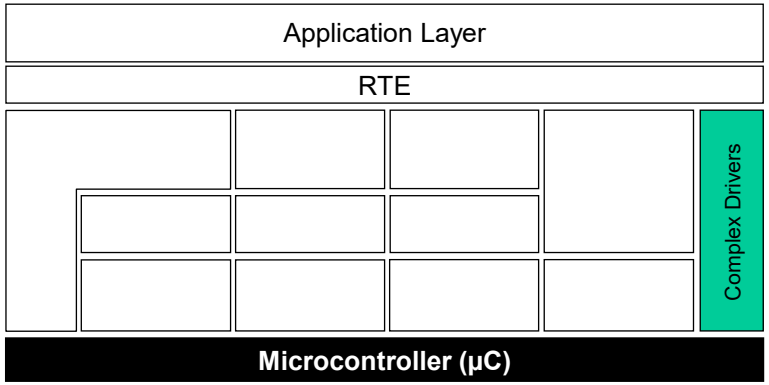


In general, it is possible to access the following modules:

- The SPI driver
- The GPT driver
- The I/O drivers with the restriction that reentrancy often only exists f 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:

- The BSW can be distributed across several cores. The core responsible for executing a call to a BSW service is determined by the task mapping of its BswOperationInvokedEvent.
- Crossing partition and core boundaries is permitted for module internal communication only, using a master/satellite implementation.
- 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 other core.
- However, if the CDD needs to access standardized interfaces of the BSW and does not reside on the same core,
  - either a satellite providing the standardized interface can run on the core where the CDD resides and forward the call to the other core
  - or a stub part of the CDD needs to be implemented on the other 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 the latter case the initialization part of the CDD also needs to reside in the stub part on the different core.

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. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. Interfaces: General Rules
    7. **Interfaces: Interaction of Layers**
    8. Overview of CP Software Clusters
2. Configuration
3. Integration and Runtime Aspects

Interfaces: Interaction of Layers – Example "Memory"
Introduction

The following pages explain using the example „memory“:

- What are the features / difference of the available memory service modules?
- 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?

Background: Comparison between memory service modules and memory types

- The different service modules (memory managers) abstract from the used non-volatile (NV) memory, but the properties of the hardware impact their design and how access is realized.
- There are constraints on the use of the different listed modules depending on the properties of the used NV hardware.
- The following table lists the properties of the modules and related NV memory.

Module	Use cases, features	Supported NV memory properties	Example hardware
NvM	• Storage of module data (e.g. Error information, special configuration info, status information, diagnostic data, ...) • Supports many reader/writer (BSW and SW-C) in parallel. • Mostly read during start-up and written in shutdown, but intermediate reads/writes during normal operation are also supported • Typical data size per user is bytes to some KiB	• Direct (memory mapped) and indirect (e.g. via SPI) NV access • Serialized access (read-while-write-in-same-HW-segment may not work → NvM always buffer the data)	• Internal data flash (via Flash EEPROM emulation) • External EEPROM / data flash
BndM	• Storage of car specific data • (Very rare) Writes via diagnostics, only in „controlled environment“ (e.g. repair shop) • Supports many readers (SW-C) in parallel • Users have direct access via pointer • Typical size many KiB	• Direct access of NV data (via pointer) is required • Parallel read of NV data is required	• Internal data flash • Internal code flash
FOTA (manager)	• Storage of model specific car data/code • Very few users, typically only one • Typical size in MiB • Write new data in the background e.g. over several driving cycles (interruptible and preemptable update procedure)	• Read-While-Write (e.g. via memory abstraction/partitioning)	• Internal and external code flash

Interfaces: Interaction of Layers – Example "Memory"
Example and First Look

This example shows how the NVRAM Manager and the Watchdog Manager interact with drivers on an assumed hardware configuration:

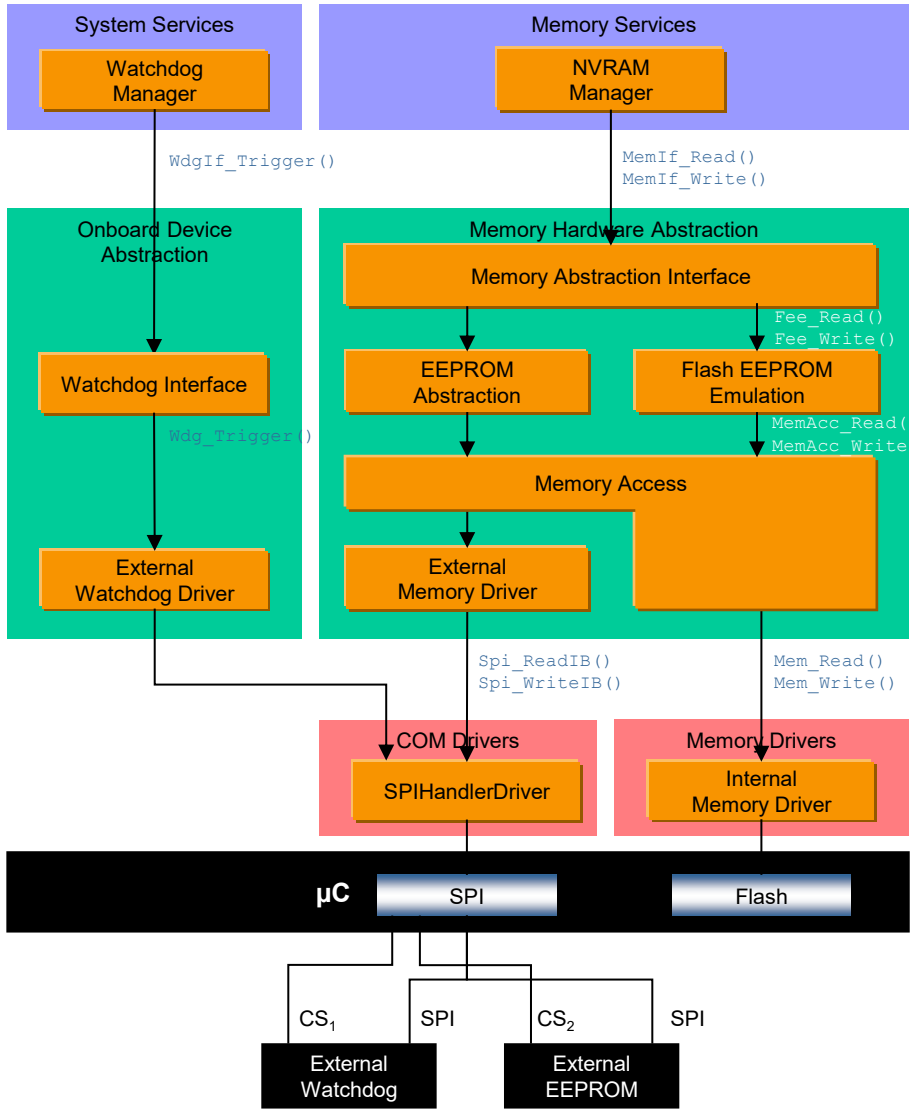
The ECU hardware includes an external EEPROM and an external watchdog connected to the microcontroller via the same SPI.

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.

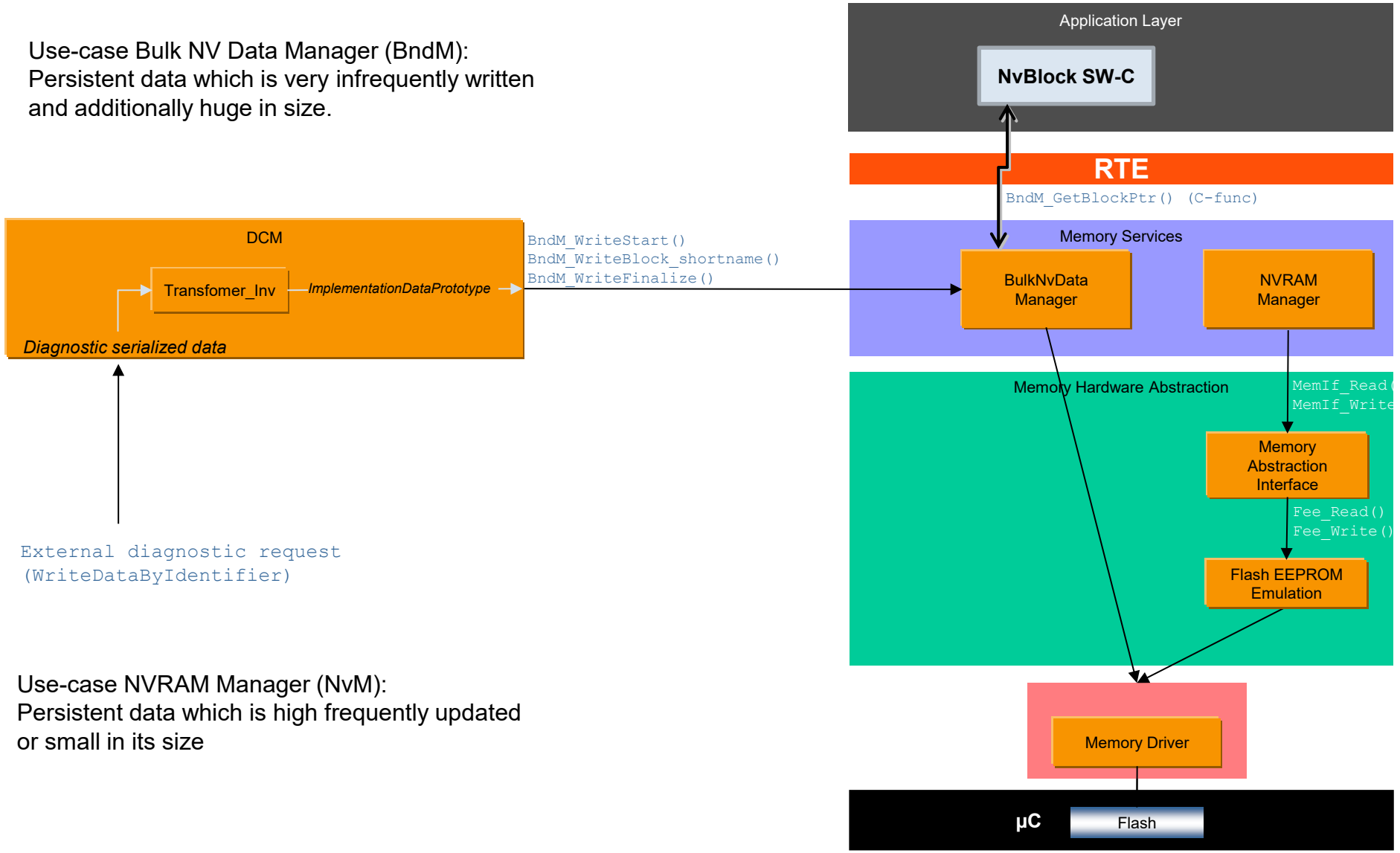
The Memory Abstraction Interface can be realized in the following ways:

- 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")



Interfaces: Interaction of Layers – Example "Memory"
Bulk NV Data Manager

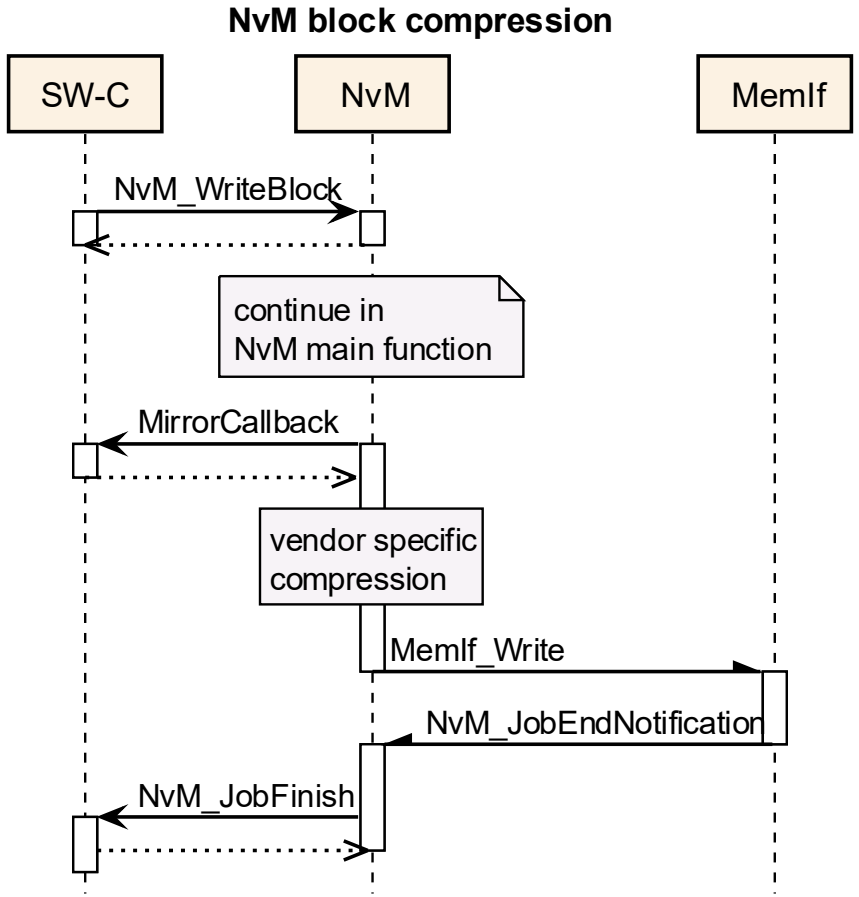
Use-case Bulk NV Data Manager (BndM):
Persistent data which is very infrequently written and additionally huge in size.



Use-case NVRAM Manager (NvM):
Persistent data which is high frequently updated or small in its size

Interfaces: Interaction of Layers – Example "Memory"
NvM Block Compression

- Use-case: large data blocks frequently written with only small local changes
  - The actual algorithm is vendor-specific (block split, compression, delta,…)



Interfaces: Interaction of Layers – Example "Memory"
Closer Look at Memory Hardware Abstraction

Architecture Description

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

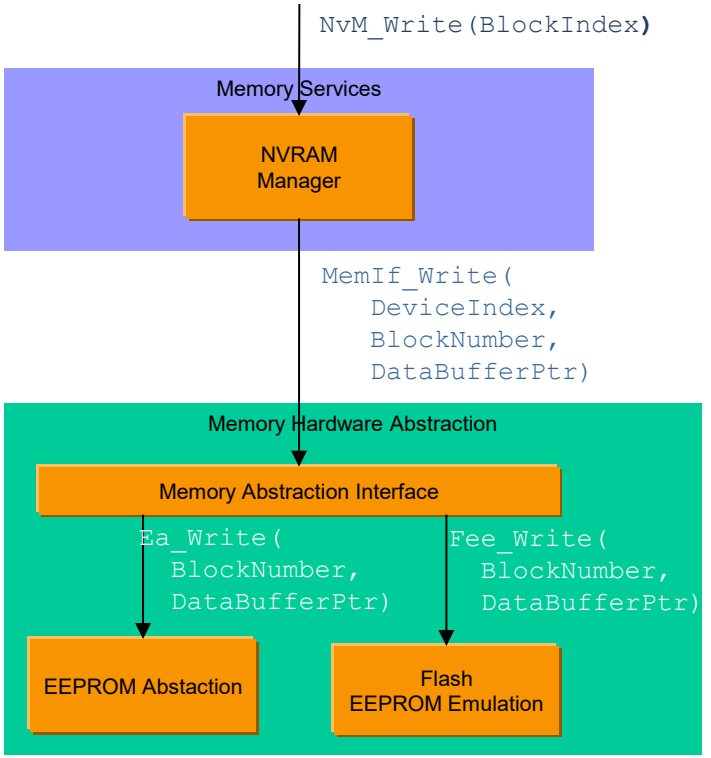
Interface Description

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

```
Std_ReturnType MemIf_Write
(
   uint8              DeviceIndex,
   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) \
  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:

```
extern const WriteFctPtrType WriteFctPtr[2];

#define MemIf_Write(DeviceIndex, BlockNumber, DataBufferPtr) \
WriteFctPtr[DeviceIndex](BlockNumber, DataBufferPtr)
```

File MemIf.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.

Interfaces: Interaction of Layers – Example "Memory"
Conclusion

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

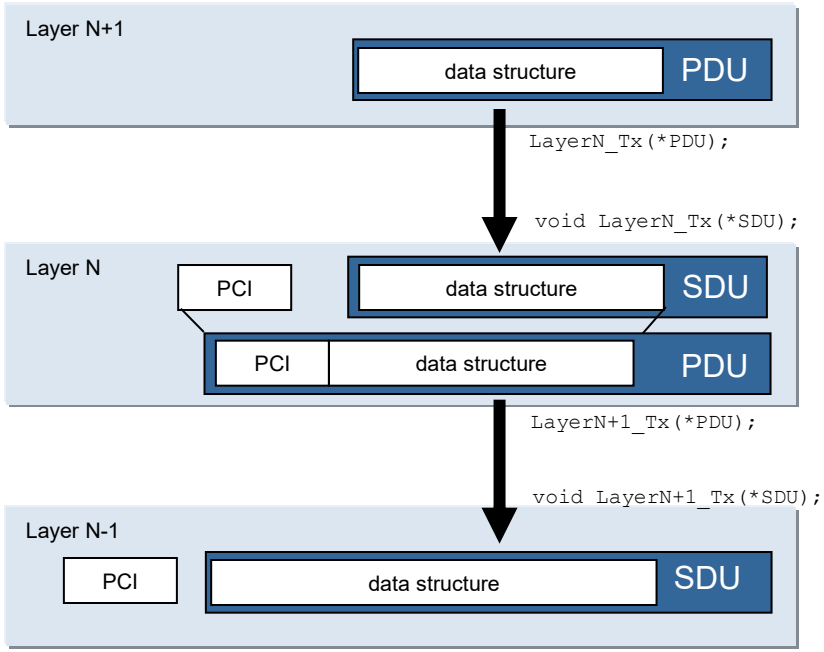
Interfaces: Interaction of Layers – Example "Communication"
PDU Flow through the Layered Architecture

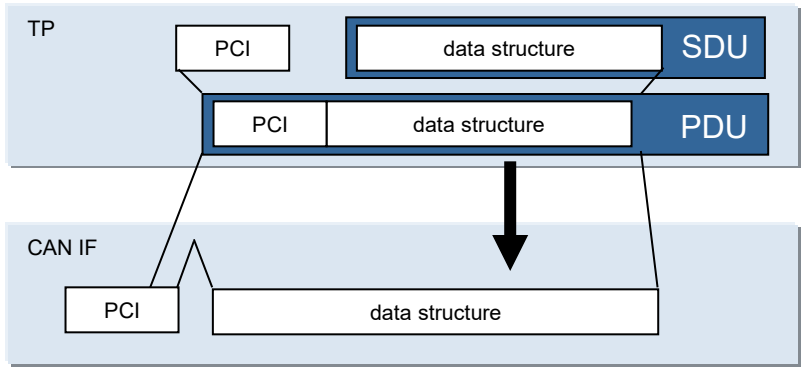
- **Explanation of terms:**

- **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.

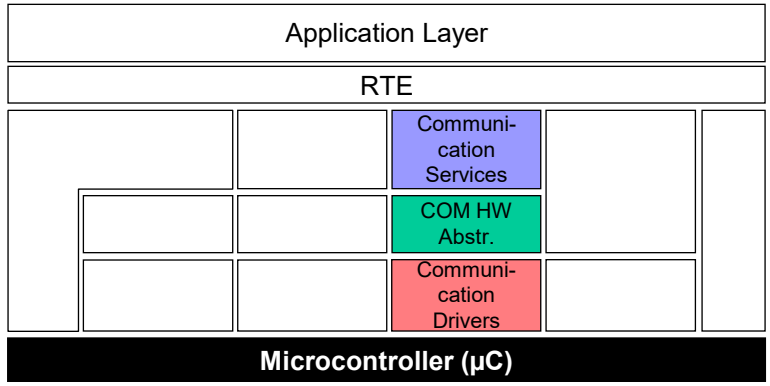
- **PCI**
  PCI is the abbreviation of "Protocol Control Information". This Information is needed to pass a SDU from one instance of a specific protocol layer to another instance. E.g. it contains source and target information.
  The PCI is added by a protocol layer on the transmission side and is removed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

ISO Layer	Layer Prefix	AUTOSAR Modules	PDU Name	CAN prefix	LIN prefix	FlexRay prefix
Layer 6: Presentation (Interaction)	I	COM, DCM	I-PDU	N/A		
	I	PDU router, PDU multiplexer	I-PDU	N/A		
Layer 3: Network Layer	N	TP Layer	N-PDU	CAN SF CAN FF CAN CF CAN FC	LIN SF LIN FF LIN CF LIN FC	FR SF FR FF FR CF FR FC
Layer 2: Data Link Layer	L	Driver, Interface	L-PDU	CAN	LIN	FR

SF: Single Frame
FF: First Frame
CF: Consecutive Frame
FC: Flow Control

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

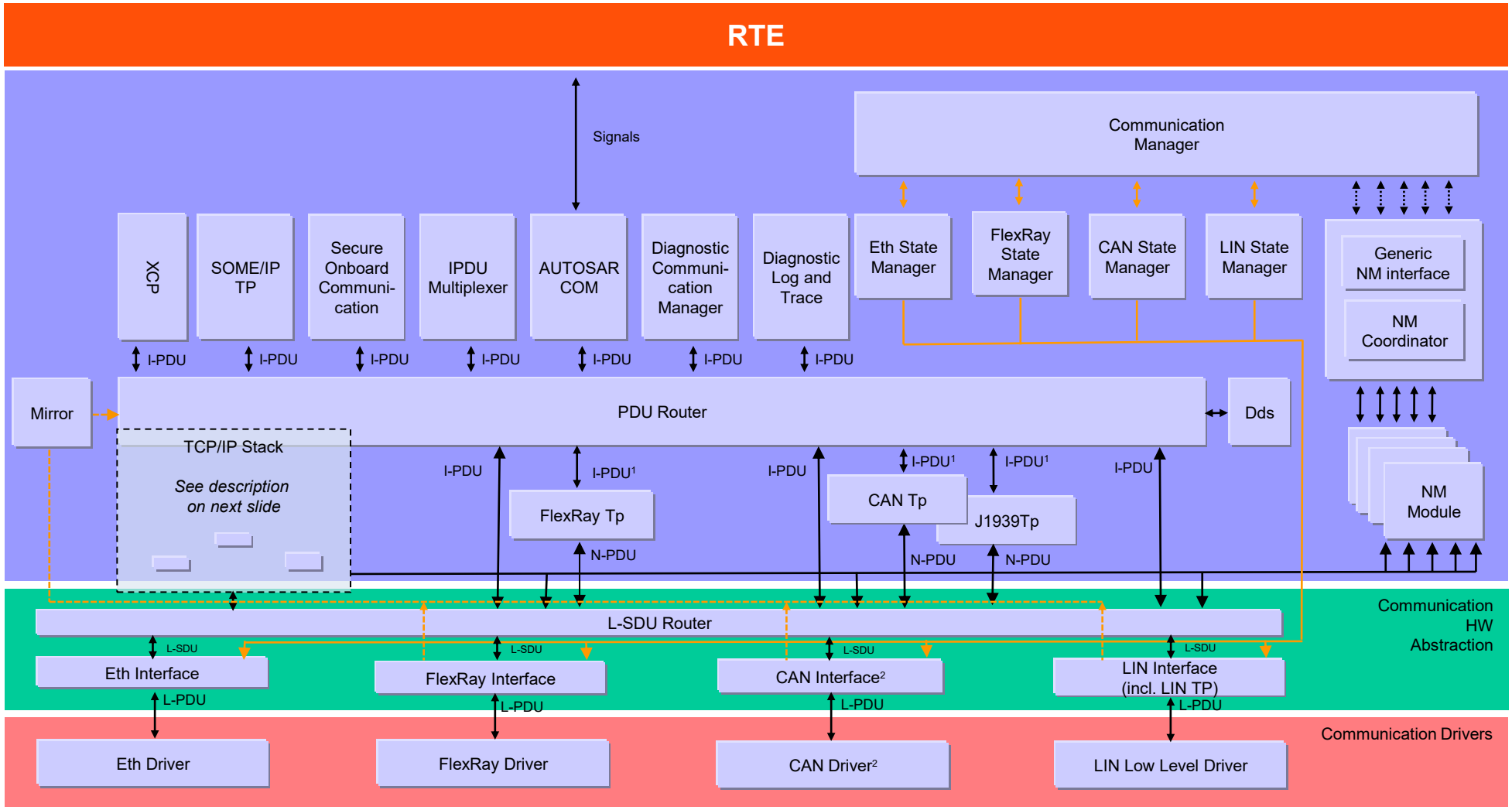
For details on the frame types, please refer to the AUTOSAR Transport Protocol specifications for CAN, LIN and FlexRay.

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)

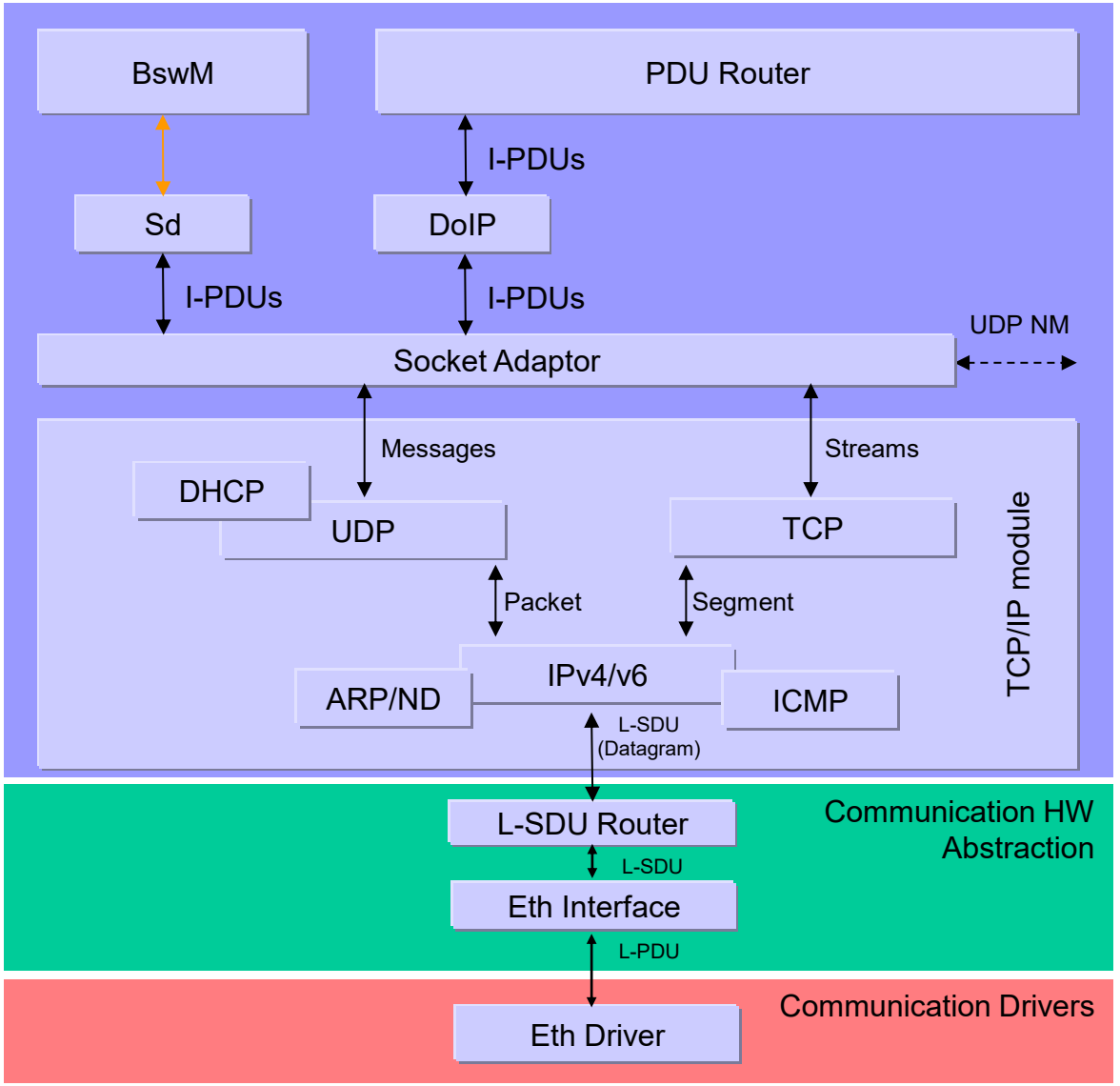


Note: This image is not complete with respect to all internal communication paths.

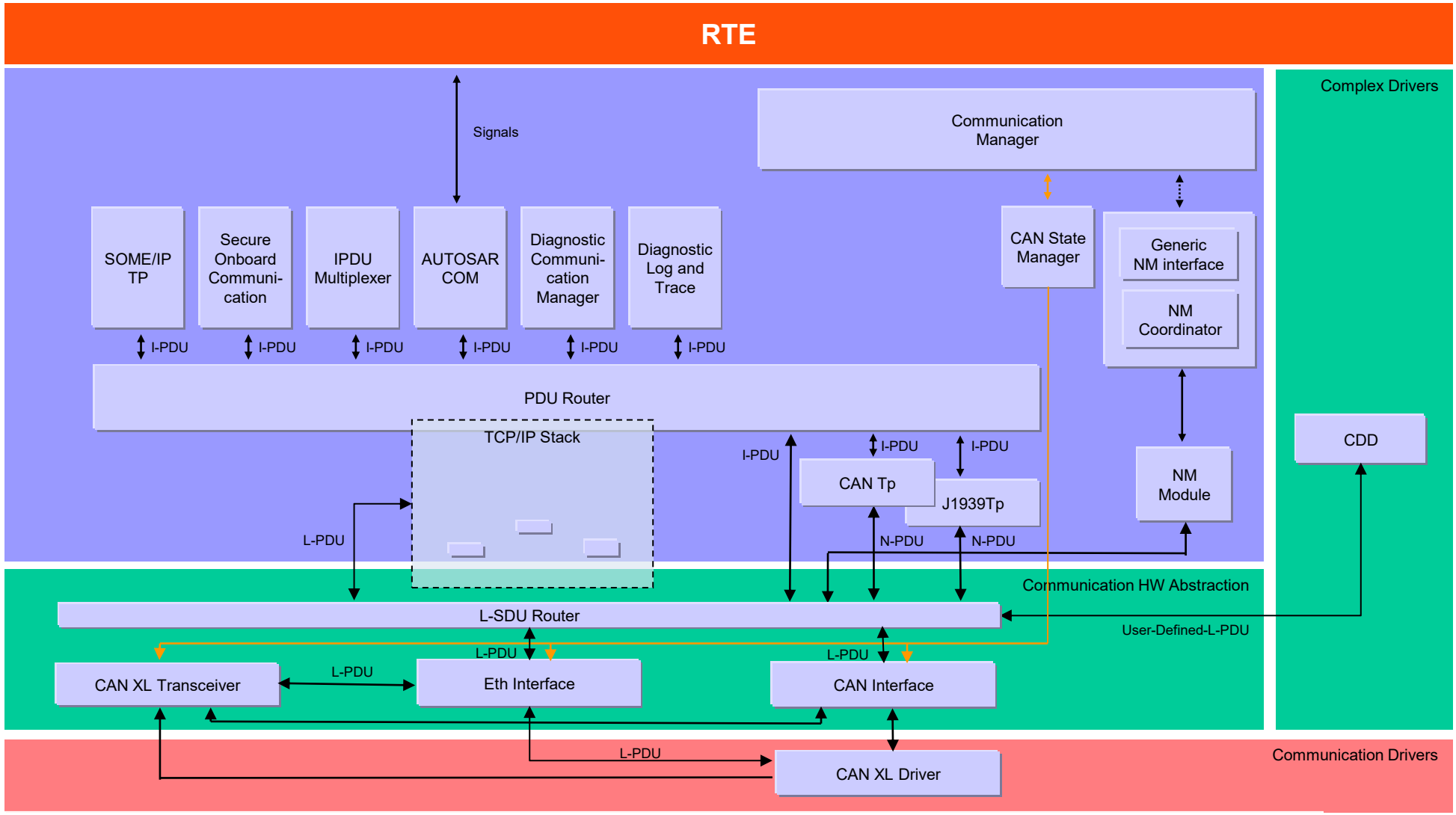
[1] The Interface between PduR and Tp differs significantly compared to the interface between PduR and the Ifs.
In case of TP involvement a handshake mechanism is implemented allowing the transmission of I-PDUs > Frame size.

Interfaces: Interaction of Layers
Example “Communication” (4) – Ethernet Stack

- This figure shows the interaction of and inside the Ethernet stack.



Interfaces: Interaction of Layers
Example “Communication” (5) - Ethernet and CAN communication using CAN XL



Note: This image is not complete with respect to all internal communication paths.

Interfaces: Interaction of Layers
Example “Data Transformation” (1) – Introduction

The following pages explain communication with Data Transformation:

- How do the software layers interact?
- How do the software interfaces look like?

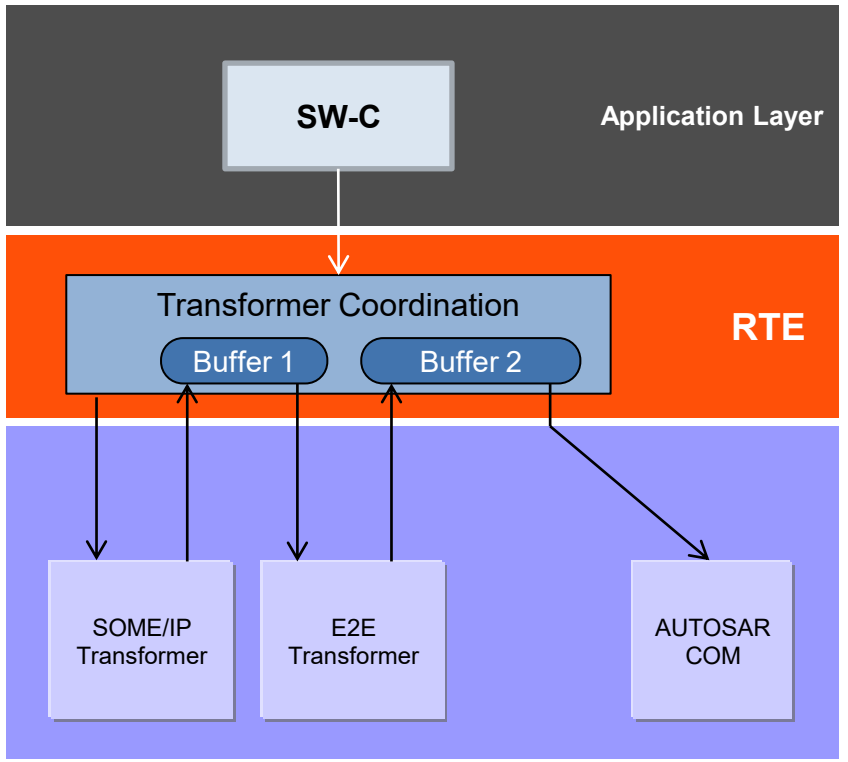
Interfaces: Interaction of Layers
Example “Data Transformation” (2) – Example and First Look

This example shows the data flow if data transformation is used for inter-ECU communication.

A SW-C sends data configured to be transmitted to a remote ECU and subject to data transformation. This data transformation doesn’t use in-place buffer handling.

Functionality

- The RTE calls the SOME/IP transformer as the first transformer in the chain and transfers the data from the SW-C.
- The SOME/IP transformer executes the transformation and writes the output (byte array) to a buffer provided by the RTE.
- Afterwards, the RTE executes the Safety transformer which is second in the transformer chain. The Safety transformer’s input is the output of the SOME/IP transformer.
- The Safety transformer protects the data and writes the output into another buffer provided by the RTE. A new buffer is required because in-place buffer handling is not used.
- The RTE transfers the final output data as a byte array to the COM module.



Interfaces: Interaction of Layers
Example “Data Transformation” (3) – Closer Look at Interfaces

Architecture Description

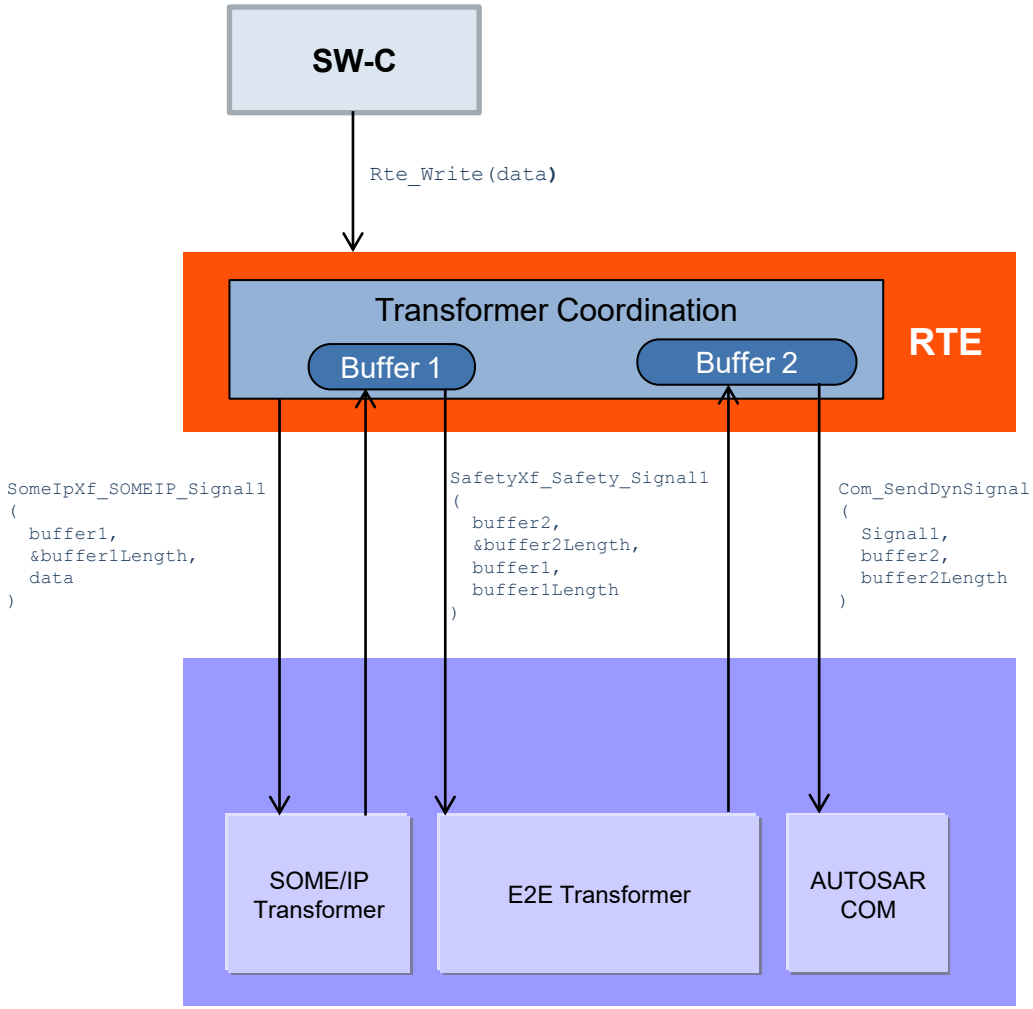
The RTE uses the transformer which are located in the System Service Layer.

Interface Description

The transformers in this example have the following interfaces:

```
SomeIpXf_SOMEIP_Signal1
(
    uint8       *buffer1,
    uint16      *buffer1Length,
    <type>      data
)

SafetyXf_Safety_Signal1
(
    uint8       *buffer2,
    uint16      *buffer2Length,
    uint8       *buffer1,
    uint16      buffer1Length
)
```



Interfaces: Interaction of Layers
Example “Data Transformation” (4) – COM Based Transformation

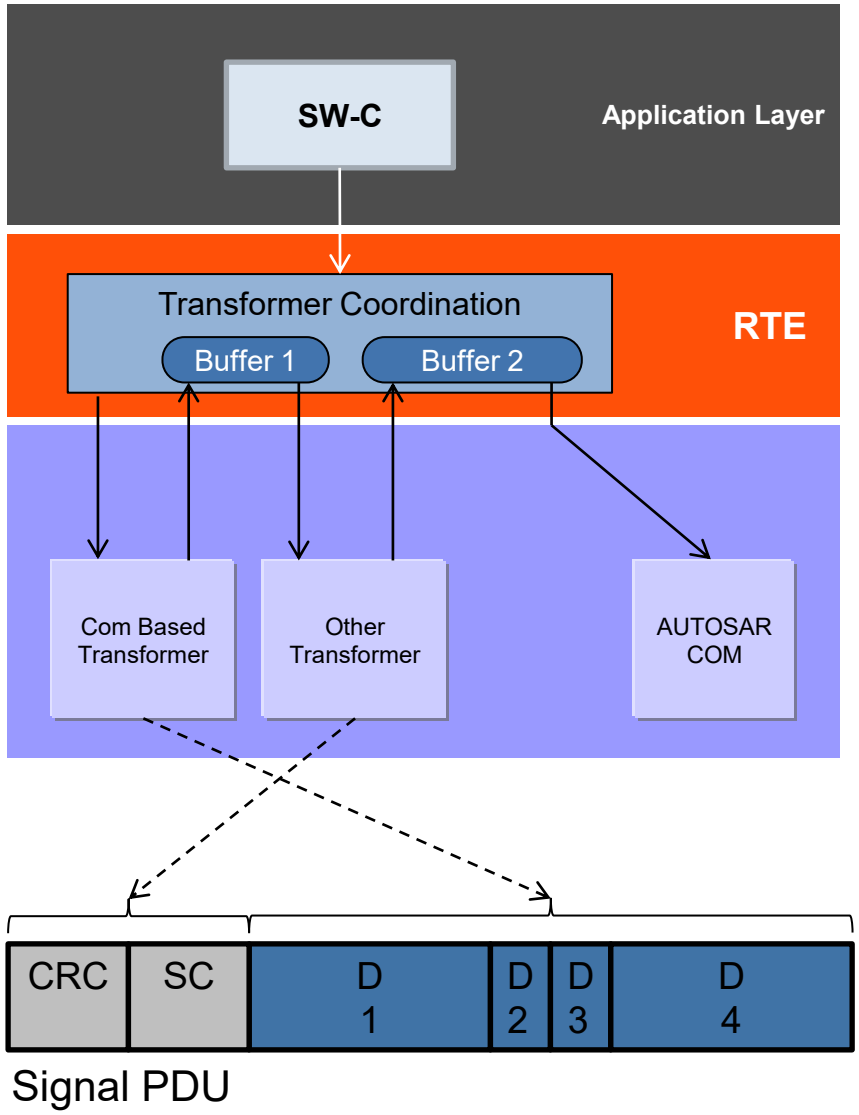
Goal

The *COM Based Transformer* provides serialization functionality to the transformer chain based on a fixed communication matrix.

The fixed communication matrix allows an optimized placement of signals into PDUs (e.g. a Boolean data can be configured to only occupy one bit in the PDU). This enables the usage of transformer chains in low payload networks like Can or Lin.

Functionality

- The *COM Based Transformer* is the first transformer (serializer) and gets the data from the application via the RTE.
- Based on the COM configuration (communication matrix) the data is serialized exactly in the same way as the COM module would have done it (endianness, sign extension).
- Other transformers may enhance the payload to have CRCs and sequence counters (SC).
- The transformer payload is passed to the COM module as one array of byte via the Com_SendSignalGroupArray API.
- The COM module can be configured to perform transmission mode selection based on the communication matrix definition.



Interfaces: Interaction of Layers
Signal-Service-Translation (1)

Goal

Adaptive Platform restricts communication to Service-oriented communication, the rest of the vehicle however still uses Signal-based communication means - therefore a translation of these two approaches has to be performed in order to allow an interaction between Classic and Adaptive Platform.

Functionality

- The definition and implementation of the Classic platform signal-service-translation shall be done inside an Application Software Component, the so called **Translation Software Component**.
- The **Translation Software Component** has Ports defined and the payload is described using **PortInterfaces**
  - Signal-to-service: Ports for incoming signals and Ports for outgoing events
  - Service-to-signal: Ports for incoming events and Ports for outgoing signals



Interfaces: Interaction of Layers
Signal-Service-Translation (2)

Functionality

- For the signal-based part the full functionality of the Classic platform **COM-Stack** is available and may be configured such that the signal-based ISignalIPdus may originate from a variety of sources (Can, Lin, FlexRay) and the ISignalIPdus may be safety and security protected.
- For the service-oriented part it has to be guaranteed that the defined SOME/IP Service actually is compatible to the Adaptive platform. This applies for the payload part (e.g. the SOME/IP serializer has to be used) as well as for the control path using BswM and ServiceDiscovery.
- The behavioral part of the Translation Software Component itself defines how the data from signal-based side is transported to the service-oriented side, and vice versa.

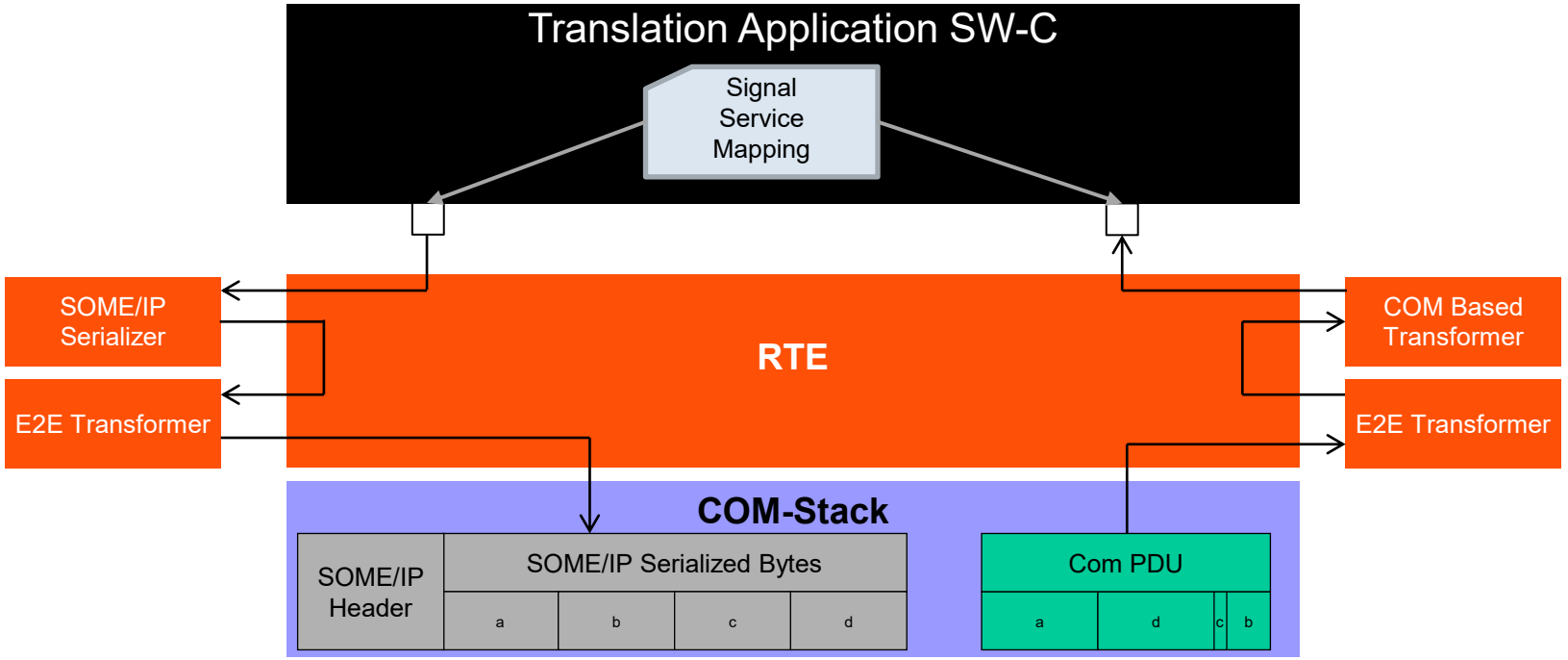
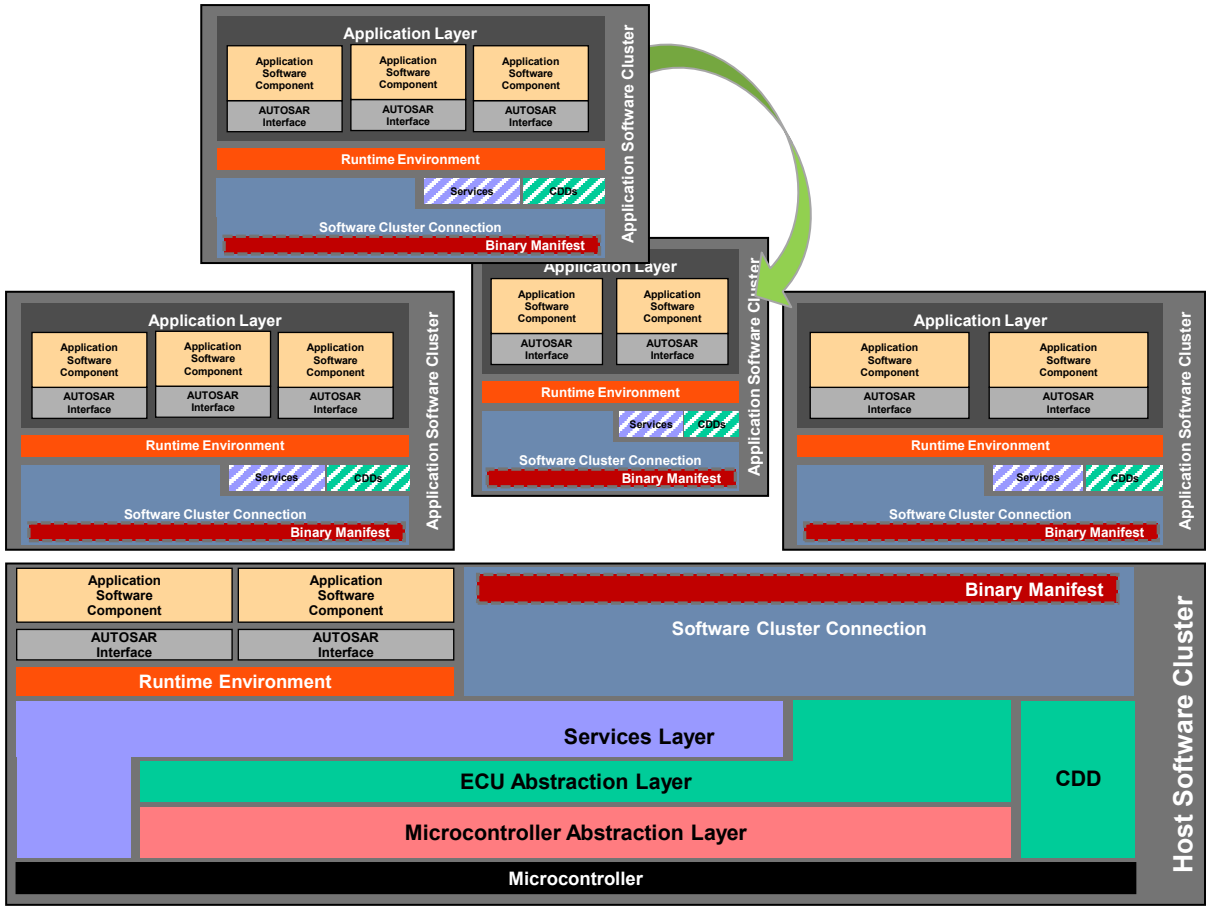


Table of contents

1. **Architecture**
    1. Overview of Software Layers
    2. Content of Software Layers
    3. Content of Software Layers in Multi-Core Systems
    4. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. Interfaces: General Rules
    7. Interfaces: Interaction of Layers
    8. **Overview of CP Software Clusters**
2. Configuration
3. Integration and Runtime Aspects

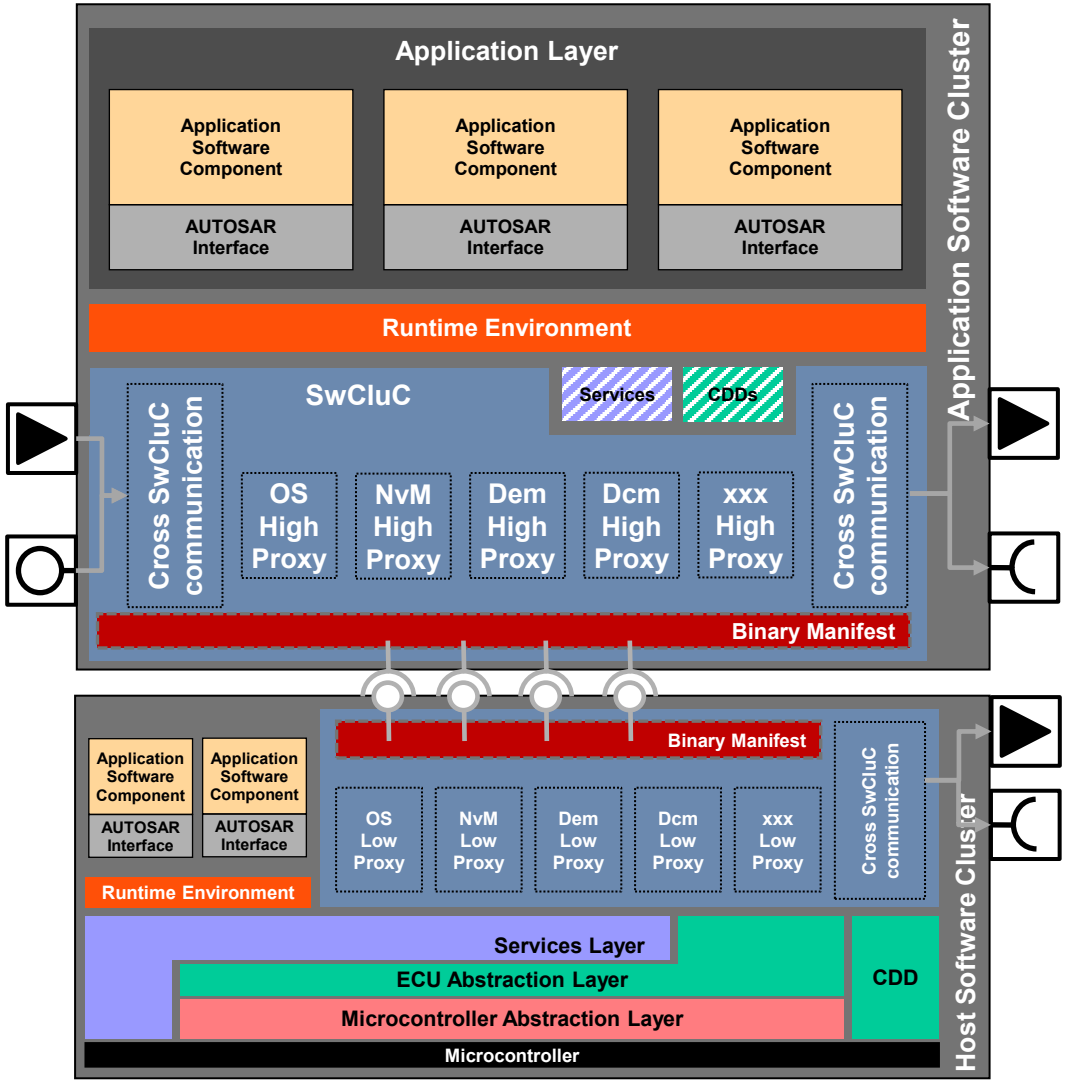
Overview of CP Software Clusters
Concept overview

The approach in a nutshell



- Software Cluster enable to split the monolithic Classic Platform Architecture into smaller units
- Each CP Software Cluster is separately buildable
- Software Clusters can be independently updated
- Connections between Software Clusters are created on basis of Binary Objects and the information hold in the Binary Manifest
- Considers the limitation of current micro controller architectures, e.g. no address virtualization
- In an *Application Software Cluster*, Application SW-Cs and BSW modules (with limitations) can be integrated
- The *Host Software Cluster* contains the major part of the BSW Stack, especially micro controller dependent modules including the **Operating System**.

Overview of CP Software Clusters
Software Cluster Connection (1)



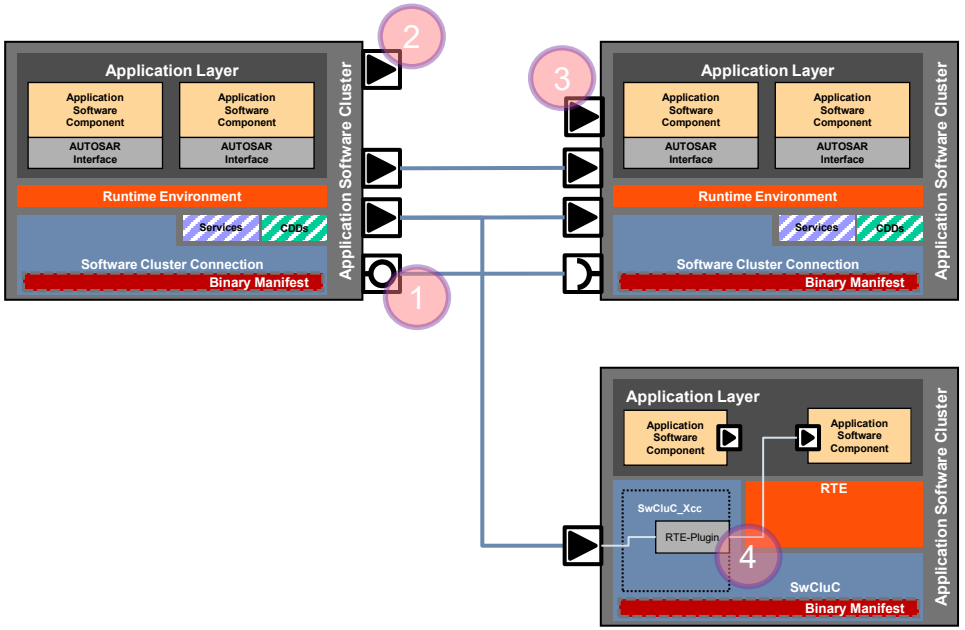
The module **Software Cluster Connection (SwCluC)** has 3 parts:

- **Cross Software Cluster Communication** (SwCluC_Xcc) provides the features in Classic Platform
  - to enable the connection of software clusters based on binary manifest
  - for cross interaction and communication of software clusters

- Abstraction of non-software cluster-local BSW modules and their APIs in the corresponding **proxy modules**
  - High Proxies substitute non-local BSW and provide the according APIs
  - Lower Proxy modules connect to regular BSW modules of the Host Software Cluster

- The **Binary Manifest** (BManif) provides binary meta information for interfaces to be able to connect software clusters.

Overview of CP Software Clusters
Software Cluster Connection (2)



- **Software Cluster Connection (SwCluC)** enables a flexible handling of interfaces
  - Interfaces will be connected in a link process, based on Binary Manifest and match of required and provided entries
  - If a match is found the connection is established (1)
  - If no requester is found the interface stays open (2)
  - If no provider is found, the interface stays open, and default values are provided (3)
  - This enables update of Software Clusters with interface changes

- **Cross Software Cluster Communication** (SwCluC_Xcc) implements the communication pattern and the interface to the RTE
  - RTE interface: RIPS-Plugin (4)

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. Content of Software Layers in Mixed-Critical Systems
    5. Overview of Modules
    6. Interfaces
        1. General
        2. Interaction of Layers (Examples)
2. **Configuration**
3. Integration and Runtime Aspects

Configuration
Overview

The AUTOSAR Basic Software supports the following configuration classes:

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

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

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

Independent of the configuration class, single or multiple configuration sets can be provided by means of variation points. In case that multiple configuration sets are provided, the actually used configuration set is to be chosen at runtime in case the variation points are bound at run-time.

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.

Note: Multiple configuration sets were modeled as a sub class of the Post-build time configuration class up to AUTOSAR 4.1.x.

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.

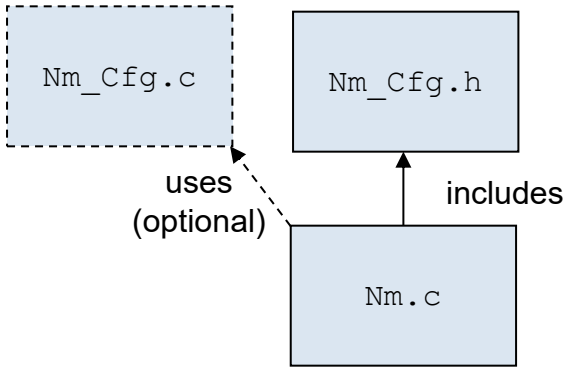
Restrictions

- The module must be available as source code
- The configuration is static and it may consist of one or more configuration sets identified by means of variation points. To update any configuration set (e.g. change the value of certain parameters), 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



Configuration
Pre-compile time (2)

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 Memory Abstraction (as specified by AUTOSAR) is not used to keep the example simple.

Configuration
Pre-compile time (3)

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):

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

#define DemConf_DemEventParameter_NVM_E_REQ_FAILED       5U
#define DemConf_DemEventParameter_CANSM_E_BUS_OFF        6U
...
```

File Dem.h:

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

File NvM.c (available as source code):

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

Dem_SetEventStatus(DemConf_DemEventParameter_NVM_E_REQ_FAILED, DEM_EVENT_STATUS_PASSED);
```

Configuration
Link time (1)

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)
- Creation of configuration 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.
2. 2..n configuration sets, runtime selection possible
   Configuration data shall be captured within external constant structs. The module gets a pointer to one of those structs at initialization time. The struct can be selected at each initialization.

Configuration
Link time (2)

Example 1: Event IDs reported to the Dem by a multiple instantiated module (example “Foo” module) only available as object code

XML configuration file of the Foo module:
Specifies that it needs the event symbol `FOO_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 */

#define DemConf_DemEventParameter_FOO_E_ERASE_FAILED_0   1U
#define DemConf_DemEventParameter_FOO_E_ERASE_FAILED_1   2U
#define DemConf_DemEventParameter_FOO_E_WRITE_FAILED_0   3U
#define DemConf_DemEventParameter_FOO_E_WRITE_FAILED_1   4U
#define DemConf_DemEventParameter_NVM_E_REQ_FAILED       5U
#define DemConf_DemEventParameter_CANSM_E_BUS_OFF        6U
...
```

File Foo_Lcfg.c:

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

const Dem_EventIdType Foo_WriteFailed[2] = {DemConf_DemEventParameter_FOO_E_WRITE_FAILED_1,
   DemConf_DemEventParameter_FOO_E_WRITE_FAILED_2};
```

File Foo.c (available as object code):

```
#include "Dem.h"       /* for reporting production errors       */
extern const Dem_EventIdType Foo_WriteFailed[];

Dem_SetEventStatus(Foo_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.

Configuration
Link time (3)

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 car versions (same application, different configuration), e.g. ECU in a low-cost car version may transmit less signals on the bus than the same ECU in a luxury car version.

Restrictions

- Implementation requires storing all possibly relevant configuration items in a flash able area and requires pointer dereferencing upon config access. Implementation precludes generation of code, which has impact on performance, code and data size.

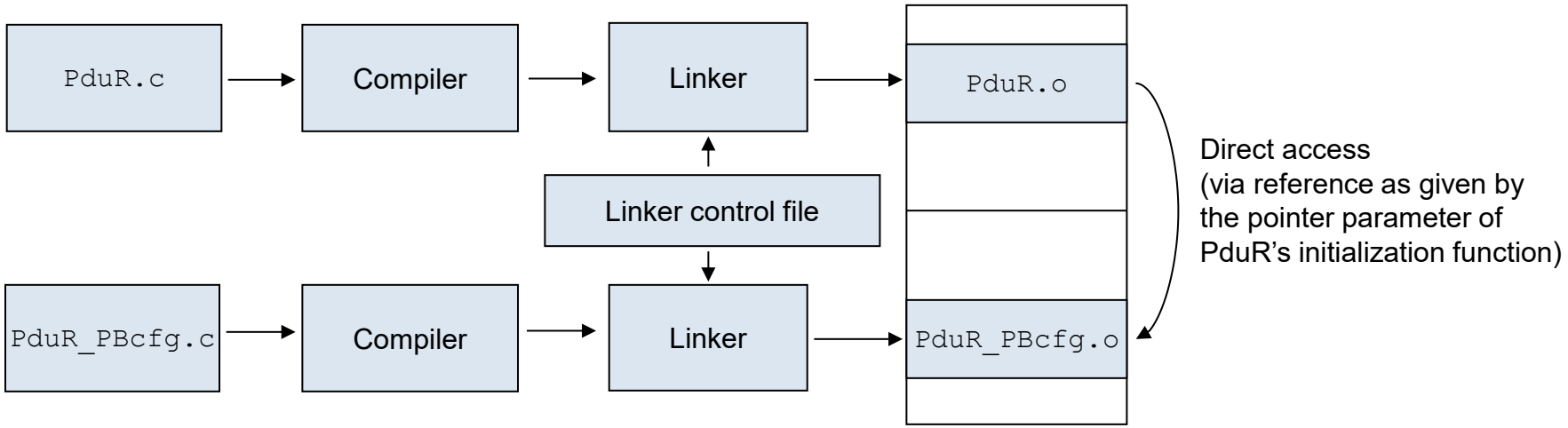
Required implementation

1. One configuration set, no runtime selection
   Configuration data shall be captured in external constant structs. These external structs are located in a separate memory segment that can be individually reloaded. The module gets a pointer to a base struct at initialization time.

2. 2..n configuration sets, runtime selection possible
   Configuration data shall be captured within external constant structs. These external structs are located in a separate memory segment that can be individually reloaded. The module gets a pointer to one of several base structs at initialization time. The struct can be selected at each initialization.

Configuration
Post-build time (2)

Example 1
If the configuration data is fix in memory size and position, the module has direct access to these external structs.



Configuration
Post-build time (3)

Required implementation 2: Configuration of CAN Driver that is available as object code only; a configuration set can be selected out of multiple configuration sets 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
       },
      {    …   }
};
```

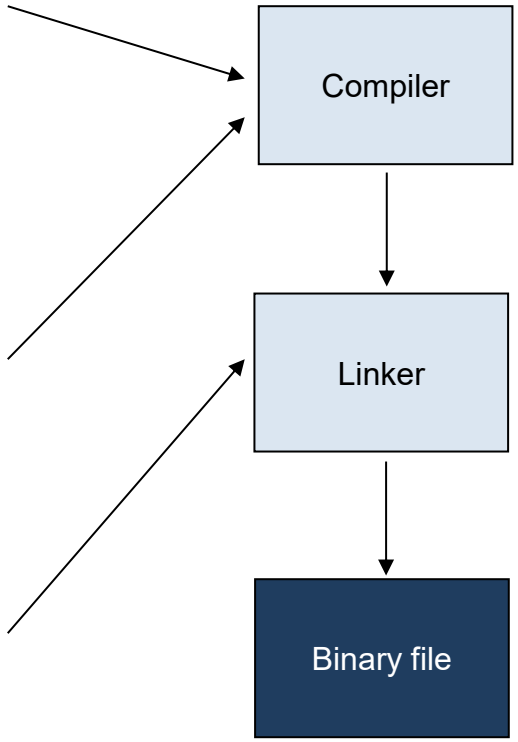
File EcuM.c:

```
#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 */
};
```



Configuration
Variants

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).

Configuration
Memory Layout Example: Post-build configuration

EcuM defines the index:

0x8000	&index (=0x8000)
0x8000	&xx_configuration = 0x4710
0x8002	&yy_configuration = 0x4720
0x8004	&zz_configuration = 0x4730
…	

Xx defines the modules configuration data:

0x4710	&the_real_xx_configuration
0x4710	lower = 2
0x4712	upper =7
0x4714	more_data
…	

Yy defines the modules configuration data:

0x4720	&the_real_yy_configuration
0x4720	Xx_data1=0815
0x4722	Yy_data2=4711
0x4724	more_data
…	

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

Configuration
Memory Layout Example: Multiple configuration sets

	0x8000	&index[] (=0x8000)
FL	0x8000	&xx_configuration = 0x4710
	0x8002	&yy_configuration = 0x4720
	0x8004	&zz_configuration = 0x4730
	…	
FR	0x8008	&xx_configuration = 0x5000
	0x800a	&yy_configuration = 0x5400
	0x800c	&zz_configuration = 0x5200
	…	
	0x8010	&xx_configuration = …
RL	0x8012	&yy_configuration = …
	0x8014	&zz_configuration = …
	…	

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 conventional single configuration case.

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. Measurement and Calibration
    7. Security
    8. Energy Management
    9. Global Time Synchronization

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, EcucPartitions, µC-Cores and BSW-Resources which have to be considered for this mapping.

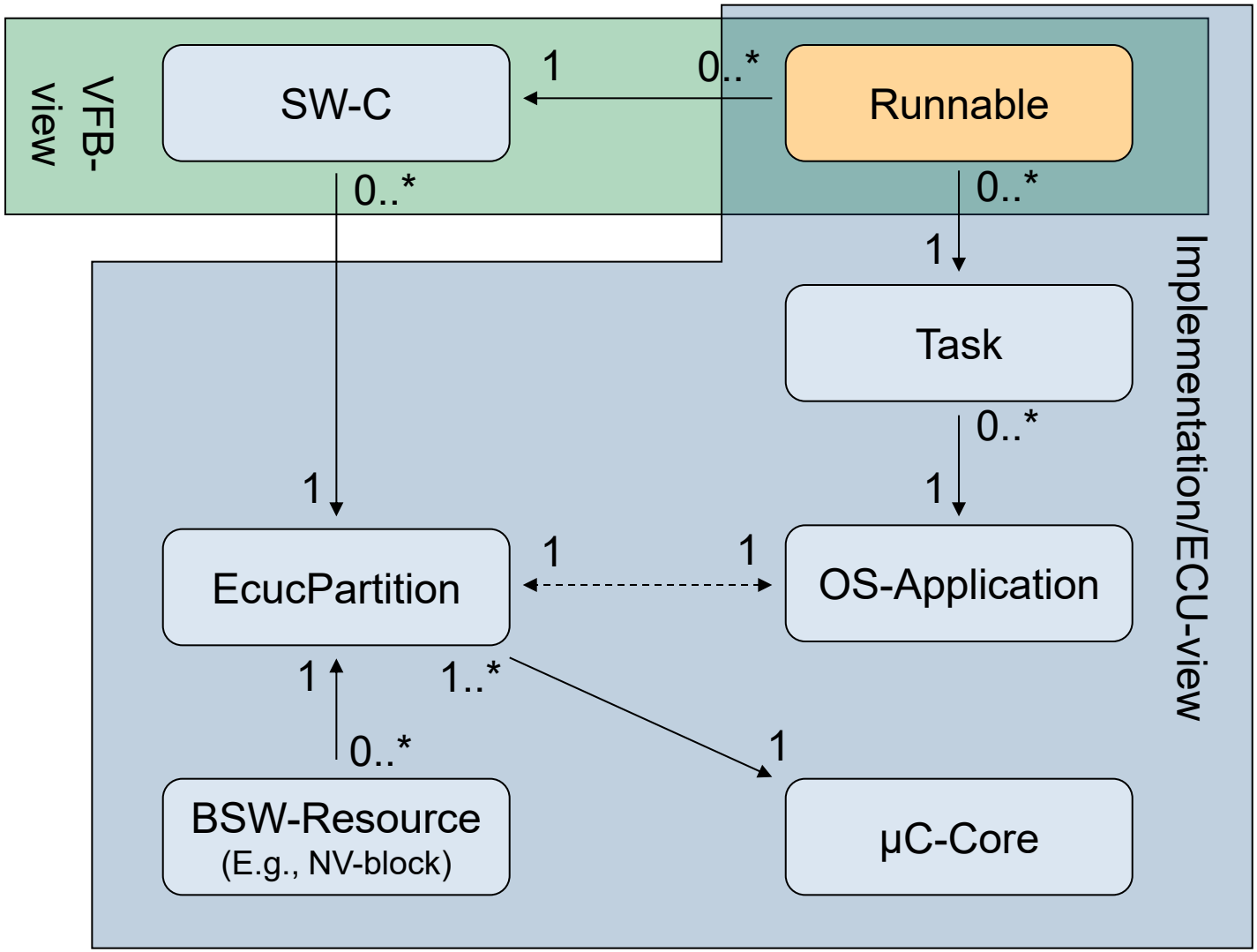


Table of contents

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. Measurement and Calibration
    7. Security
    8. Energy Management
    9. Global Time Synchronization

Integration and Runtime Aspects - Partitioning
Introduction

- Partitioning is implemented by using OS-Applications within the OS

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

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

- OS-Applications can be terminated during run-time as a result of a detected error:
  - All BSW modules are placed in privileged OS-Applications
  - These OS-Applications should not be terminated

- OS-Applications are configured in the ECU configuration:
  - SW-Cs are indirectly mapped to OS-Applications (see also slide 11eer, consequence: restricts runnable to task mapping)

- Communication across OS-Application boundaries is realized by the IOC

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. Measurement and Calibration
    7. Security
    8. Energy Management
    9. Global Time Synchronization

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 `GetElapsedCounterValue` 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

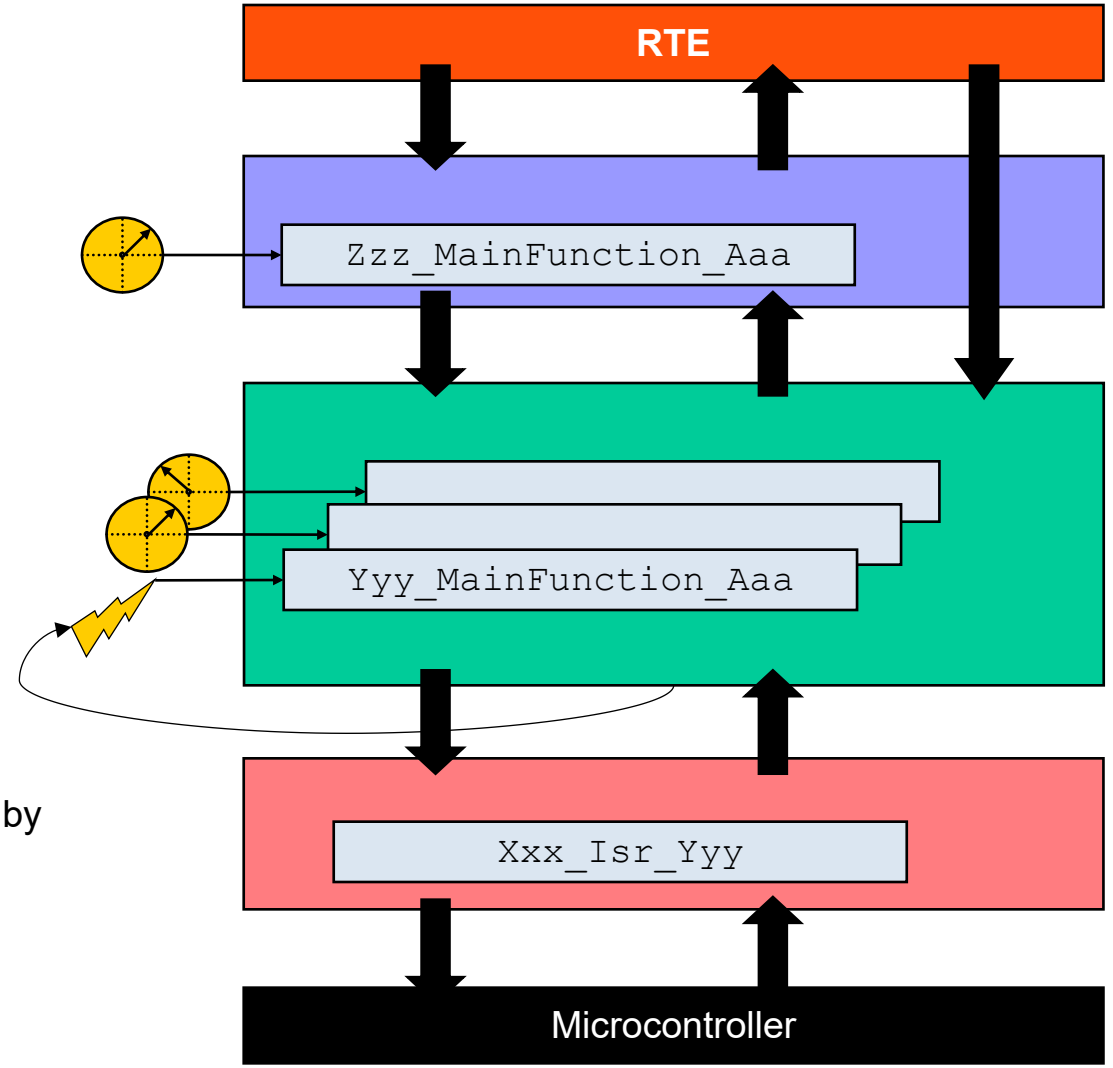
- `BswTimingEvent`
- `BswBackgroundEvent`
- `BswModeSwitchEvent`
- `BswModeSwitchedAckEvent`
- `BswInternalTriggerOccuredEvent`
- `BswExternalTriggerOccuredEvent`
- `BswOperationInvokedEvent`

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



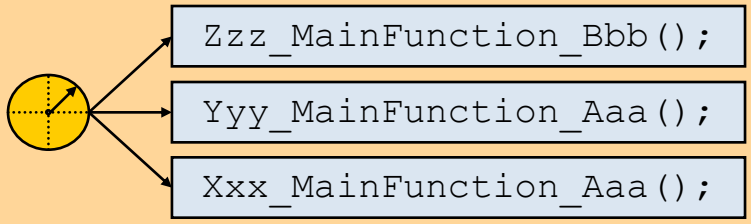
- OS objects
  - Tasks
  - ISRs
  - Alarms
  - Resources
  - OS services
- Sequences of scheduling objects within tasks
- Sequences of tasks
- ...



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

Integration and Runtime Aspects - Scheduling
Transformation Process – Example 1

Logical Architecture (Model)



Technical Architecture (Schedule Module)

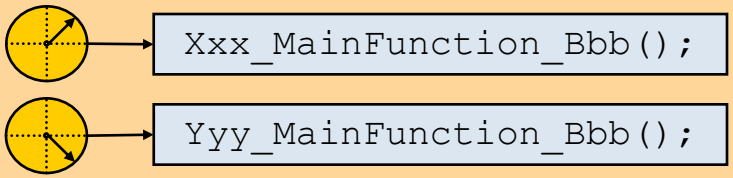
```
Task1 {
  Zzz_MainFunction_Bbb();
  Yyy_MainFunction_Aaa();
  glue code
  Xxx_MainFunction_Aaa();
  glue code
  ...
}
```

Transformation

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

Integration and Runtime Aspects - Scheduling
Transformation Process – Example 2

Logical Architecture (Model)



Technical Architecture (Schedule Module)

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

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

Transformation

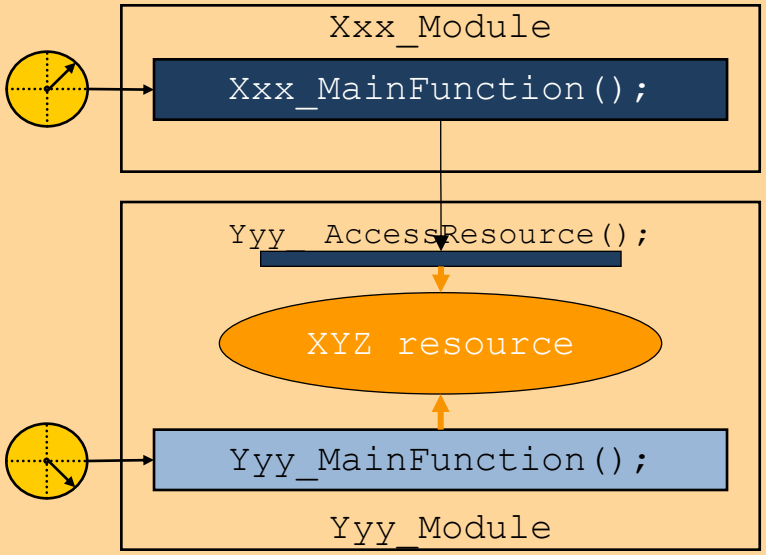
- Mapping of scheduling objects to OS Tasks

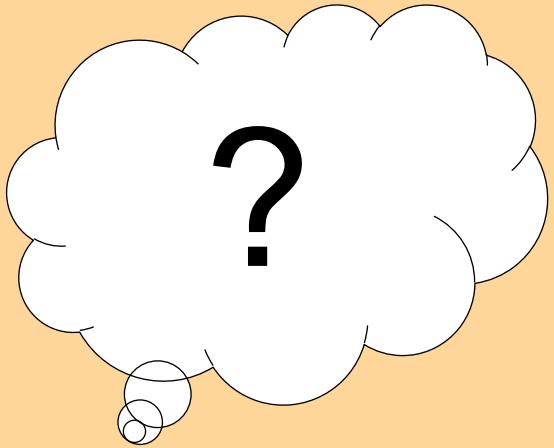
Integration and Runtime Aspects - Scheduling
Data Consistency – Motivation

Logical Architecture (Model)

Technical Architecture (Schedule Module)

- 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)





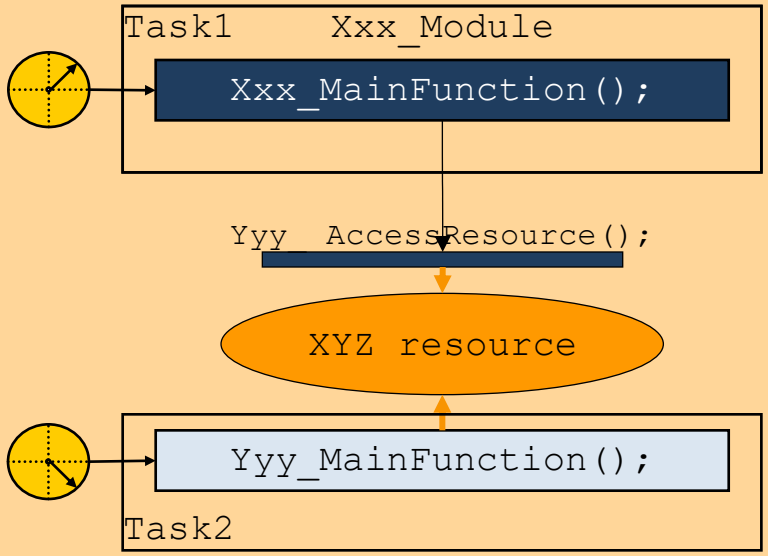
Transformation

Data consistency strategy to be used:

- Sequence, Interrupt blocking, Cooperative Behavior, Semaphores (OSEK Resources), Copies of ...

Integration and Runtime Aspects - Scheduling
Data Consistency – Example 1 – "Critical Sections" Approach

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



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();
    ...
}
```

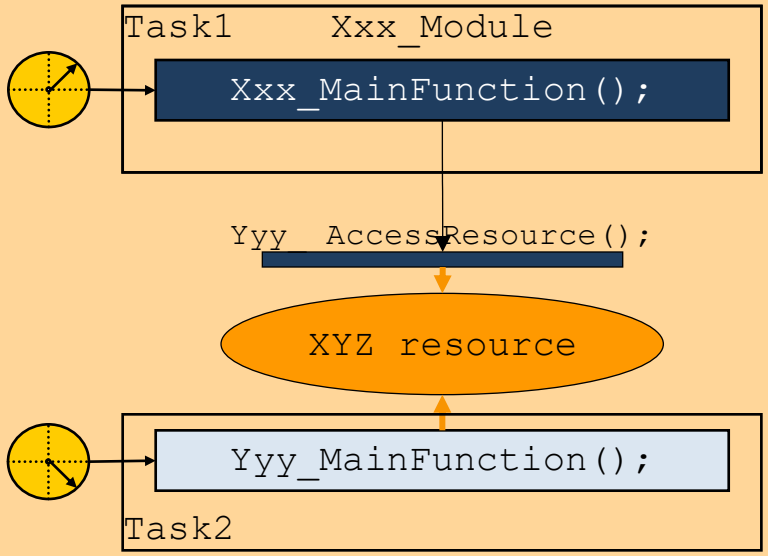
Transformation

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)



Implementation of Schedule Module

```
#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();
    ...
}
```

Transformation

Data consistency is ensured by:

- Sequence

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 `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 `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.

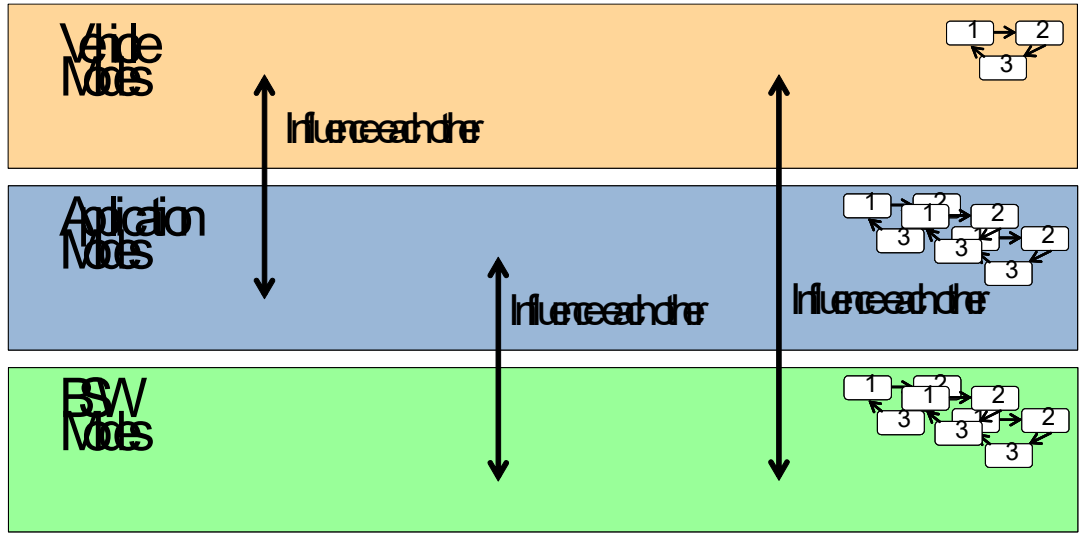
Table of contents

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. Measurement and Calibration
    7. Security
    8. Energy Management
    9. Global Time Synchronization

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 this 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.

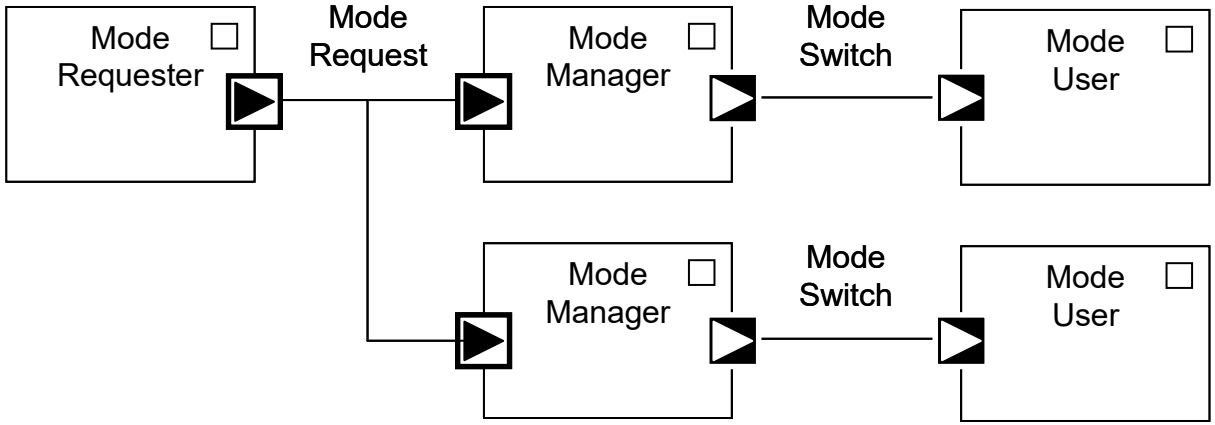
Vehicle and application mode management (2)

Processing of Mode Requests

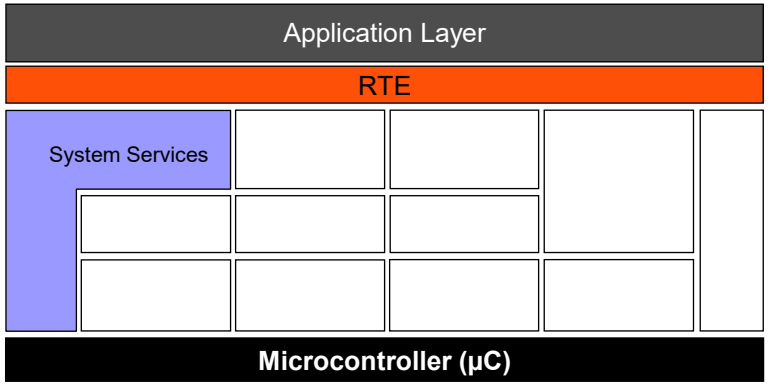
The basic idea of vehicle mode management is to distribute and arbitrate mode requests and to control the BSW locally based on the results.

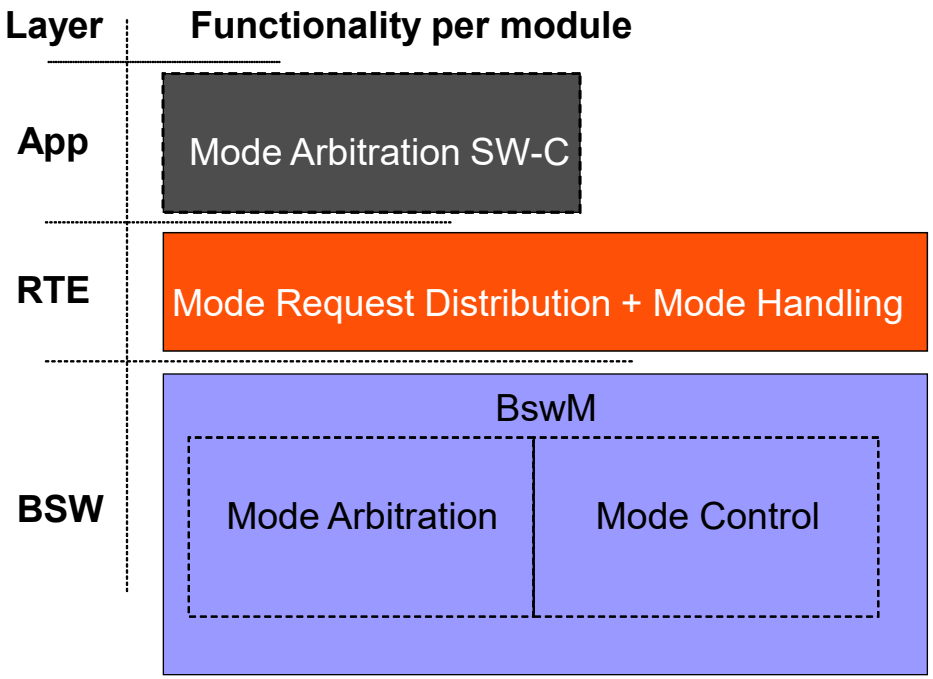
This implies that in each OS-Application, there has to be a mode manager that switches the modes for its local mode users and controls the BSW. Of course there can also be multiple mode managers that switch different Modes.

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 the Basic Software Mode Manager (BswM for short). Since the BswM is located in the BSW, it is present in every OS-Application 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.
- BswMs running in different OS-Applications can propagate mode requests by Sender-Receiver communication (SchM_Send, SchM_Receive).

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 or OS-Applications.
- 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 OS-Applications.

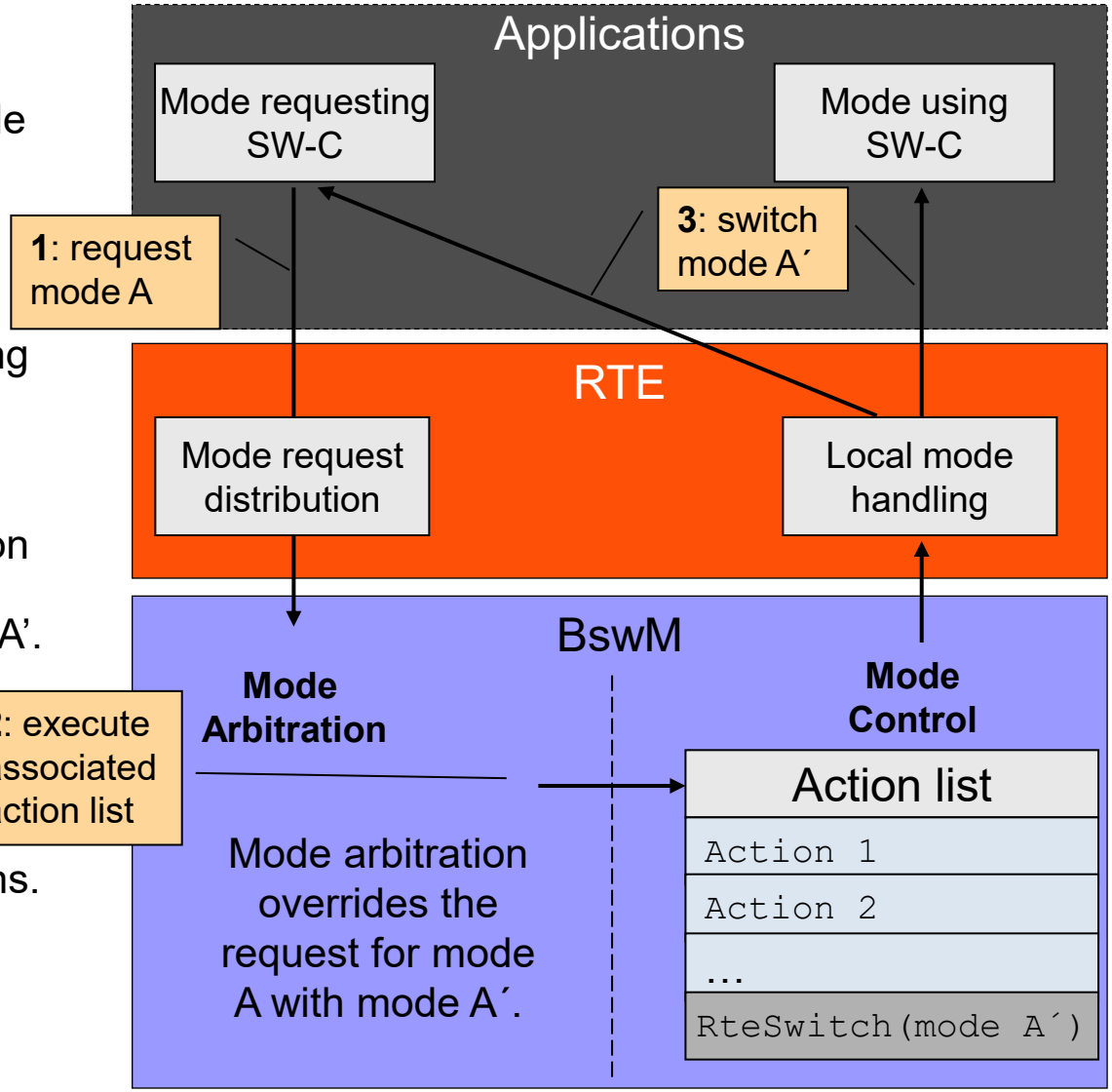


Table of contents

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. Measurement and Calibration
    7. Security
    8. Energy Management
    9. Global Time Synchronization

Error Classification (1)

Types of errors

Hardware errors / failures

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

Software errors

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

System errors

- Example 1: CAN receive buffer overflow
- Example 2: time-out for receive messages

Error Reporting – Alternatives

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

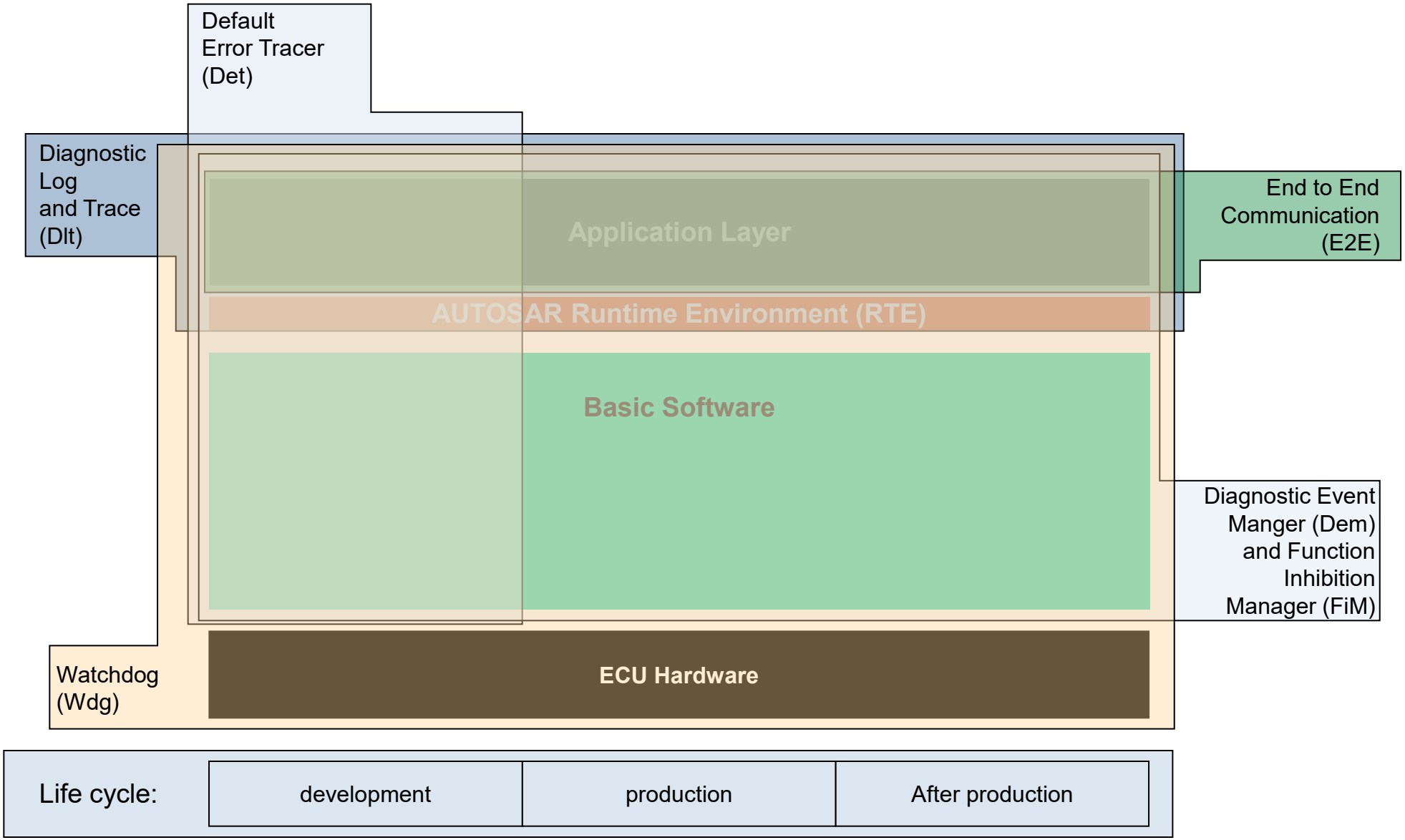
Via central Error Hooks (Default Error Tracer, Det)
For logging and tracing errors during product development. Can be switched off for production code.

Via central Callouts (Default Error Tracer, Det)
For handling errors during product life time.

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 Diagnostic Event Manager (Dem).

Integration and Runtime Aspects - Error Handling, Reporting and Diagnostic Mechanism in relation to AUTOSAR layers and system life time



Error Classification (2)

Error Classes

- Development Errors

  Development errors are software errors. They shall be detected like assertions and fixed during development phase. The detection of errors that shall only occur during development can be switched off per module for production code (by static configuration namely preprocessor switches). The according API is specified within AUTOSAR, but the functionality can be chosen/implemented by the developer according to specific needs.

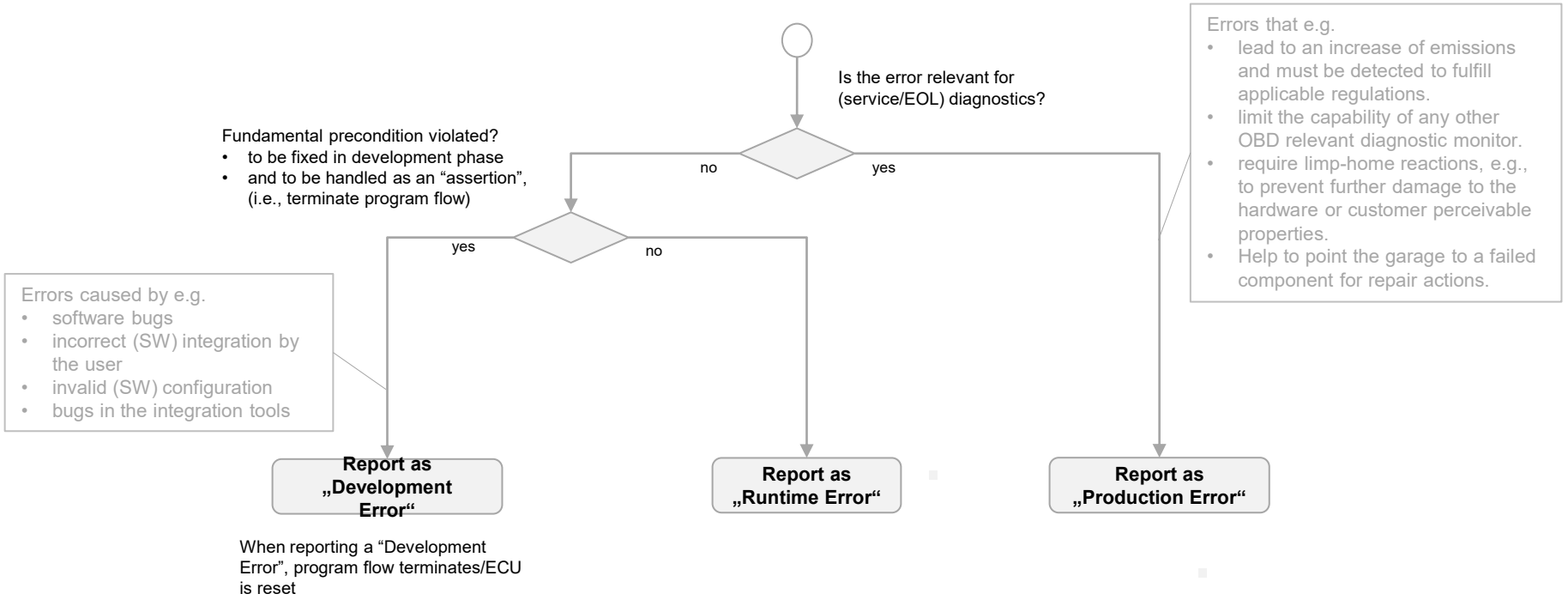
- Runtime Errors

  Runtime errors are systematic software errors. They indicate severe exceptions that hinder correct execution of the code. The monitors may stay in code even for a deployed systems. Synchronous handling of these errors can be done optionally in integrator code.

- Production Errors

  Those errors are stored in fault memory for repair actions in garages. Their occurrence can be anticipated and cannot be avoided in production code. Production errors have a detection and a healing condition.

Integration and Runtime Aspects - Error Handling, Reporting and Diagnostic
Error Classification (3)



Integration and Runtime Aspects - Error Handling, Reporting and Diagnostic
Error Classification Summary

Error Type	Main purpose	Detection and reporting	Configuration	Detection mechanism	Program flow	Reported to
Error Return Code	Inform API caller about current state of an error Leave error reaction to calling module, which has more context to decide on proper error reaction	During all life cycles	none	Event based (set condition only)	continues	To calling BSW module (synchronuously via API return code or asynchronuously via notification callback)
Development Error	Detect SW/integration issues early during development.	During „Develop-ment“ only	Module specific overall enabling/disabling of Detection + Reporting of development errors via pre-compile switch DevErrorDetect provides Callout for integration code in Det_ReportError	Event based (set condition only)	terminates	DET (via Det_ReportError; integration code in Callout allows some preparation before ECU is reset)
Runtime Error	Detect errors during normal operation, e.g. systematic software faults, timing faults or hardware issues	During all life cycles	Always active provides Callout for integration code in Det_ReportRuntimeError	Event based (set condition only)	continues	DET (via Det_ReportRuntimeError; integration code in Callout allows further reporting)
Production Error	Detect issues (and absense of issues) relevant e.g. for service/repair actions or system degradation	During all life cycles	<Ma>DemEventParameterRefs parameter allows to enable/disable the error	Maturing/healing (set/reset conditions)	continues	DEM (via Dem_SetEventStatus to the event memory)

Error Reporting via API

Error reporting via API

Informs the caller about failure of an operation by returning an error status.

Basic return type

Success: `E_OK` (value: 0)

Failure: `E_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 Default Error Tracer (Det).

Example: services of ADC driver

Success: `ADC_E_OK`

General error (service not accepted): `ADC_E_NOT_OK`

ADC is busy with current conversion: `ADC_E_BUSY`

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 Default Error Tracer (Det)

For reporting development/runtime errors.

The Default Error Tracer is mainly intended for handling errors during development time but also for handling systematic errors in production code. Within the Default 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 detection and reporting of development errors to the Default Error Tracer can be statically switched on/off per module (preprocessor switch or different object code builds of the module) but not for Runtime errors.

Integration and Runtime Aspects - Error Handling, Reporting and Diagnostic
Diagnostic Event Manager – Diagnostic Error Reporting

API

The **Diagnostic Event Manager** has the following API:

```
Dem_SetEventStatus(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
Default Error Tracer – Example: Development Error Reporting

API

The **Default Error Tracer** has the following API for reporting development errors (runtime errors use identical APIs with different names):

```
Det_ReportError(uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId)
```

Error numbering concept

`ModuleId (uint16)`

The Module ID contains the AUTOSAR module ID from the CP SWS BSWGeneral.

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.

Diagnostic Log and Trace (1)

The module **Diagnostic Log and Trace (Dlt)** collects log messages and converts them into a standardized format. The Dlt module forwards the data to the PduR, which sends it to the configured communications bus.

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 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.

Diagnostic Log and Trace (2)

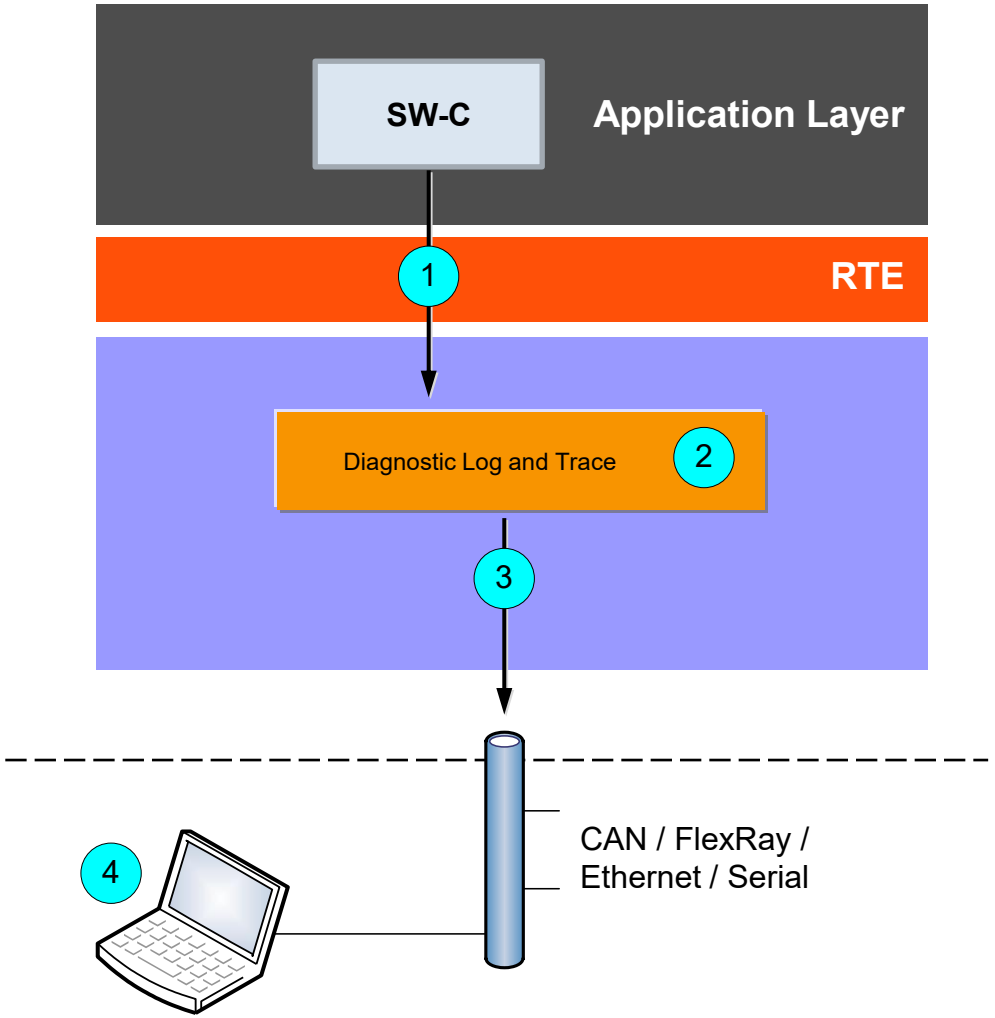
The Dlt communication module is enabled by an external client.

(1) A **SW-C** is generating a log message. The log message is sent to Dlt by calling the Interface provided by Dlt

(2) Dlt implements the Dlt protocol

(3) Dlt sends the encoded log message to the communication bus

(4) An external Dlt client collects the log message and provides it for later analysis



Diagnostic Log and Trace (3)

API

The **Diagnostic Log and Trace** has syntactically the following API:

```
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

Messaged 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`

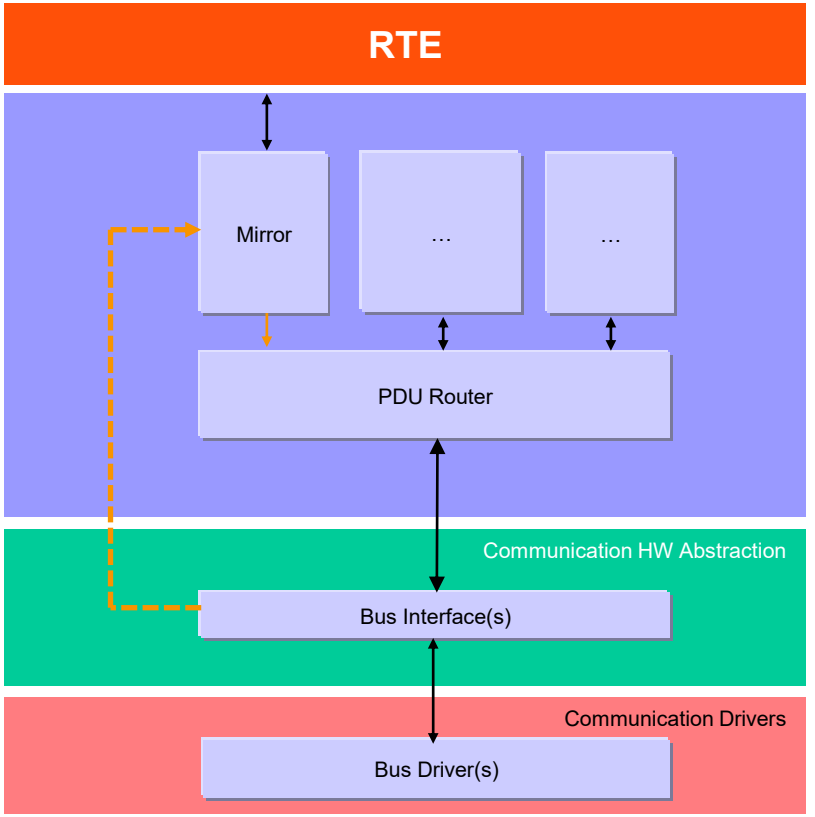
Contain the log or trace data it self. The content and the structure of this provided buffer is specified by the Dlt transmission protocol.

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.

Integration and Runtime Aspects - Error Handling, Reporting and Diagnostic
Mirroring bus communication

- The Bus Mirroring module (Mirror) collects frames from several buses and forwards these to a common destination bus, either one by one or collected into a stream.
- The Mirror module defines the protocol to be used inside the stream of mirrored frames.
- The Mirror module shall only allow for uni-directional forwarding to avoid vulnerability of the internal networks from an external link (that needs to be configured as destination of mirroring).



Integration and Runtime Aspects - Error Handling, Reporting and Diagnostic
Data collection during operation: VDP

VDP is the **Vehicle Data Protocol** of AUTOSAR which is realized by the **VdpCmR** module.

VDP is an AUTOSAR standard for the purpose of enabling ECU data collection in production vehicles.

VDP within AUTOSAR provides the following features

- A systematic decoupling of data acquisition (sampling) in the remote ECU and data transmission.
- Dynamic configuration of which data shall be collected.
- Activation and deactivation of data points to stabilize temporary pausing of data collection.

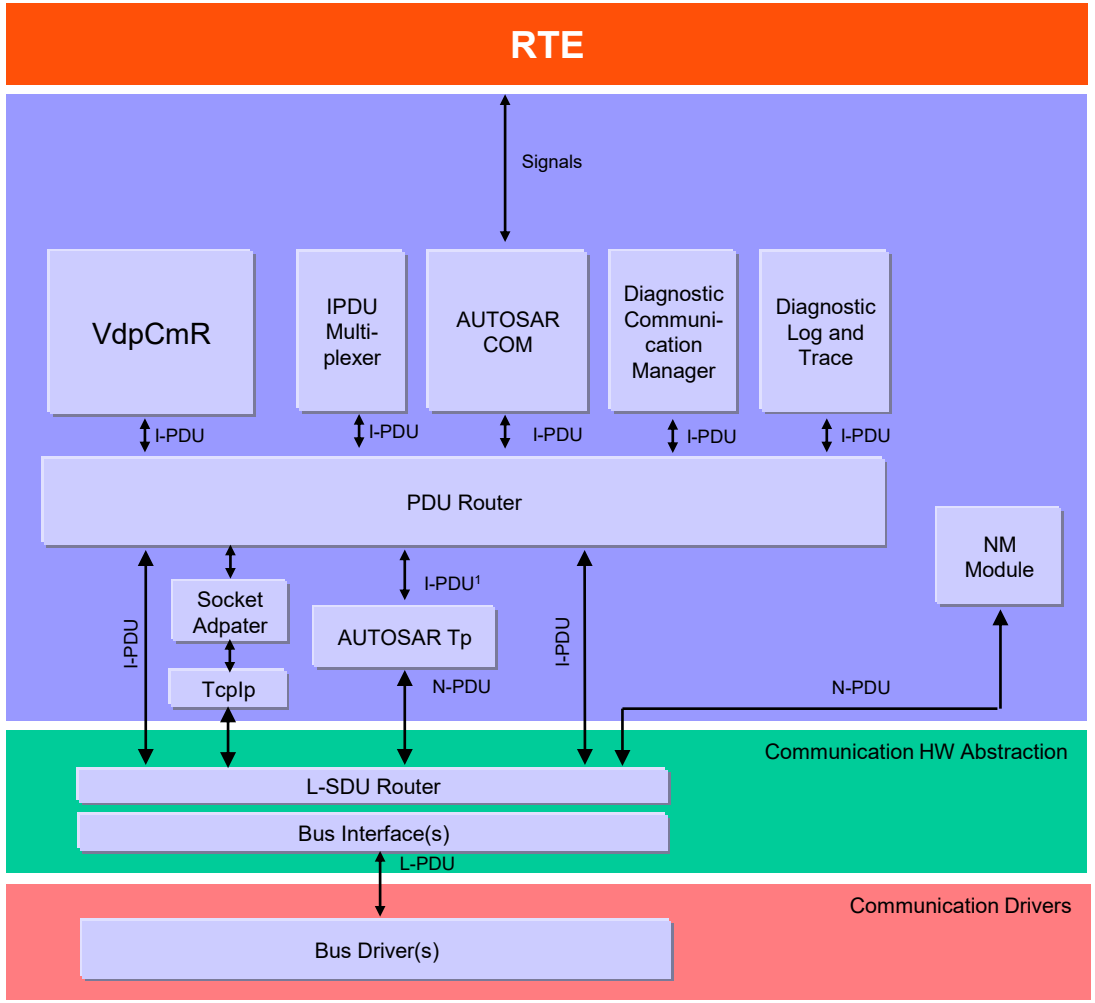


Table of contents

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. **Measurement and Calibration**
    7. Security
    8. Energy Management
    9. Global Time Synchronization

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

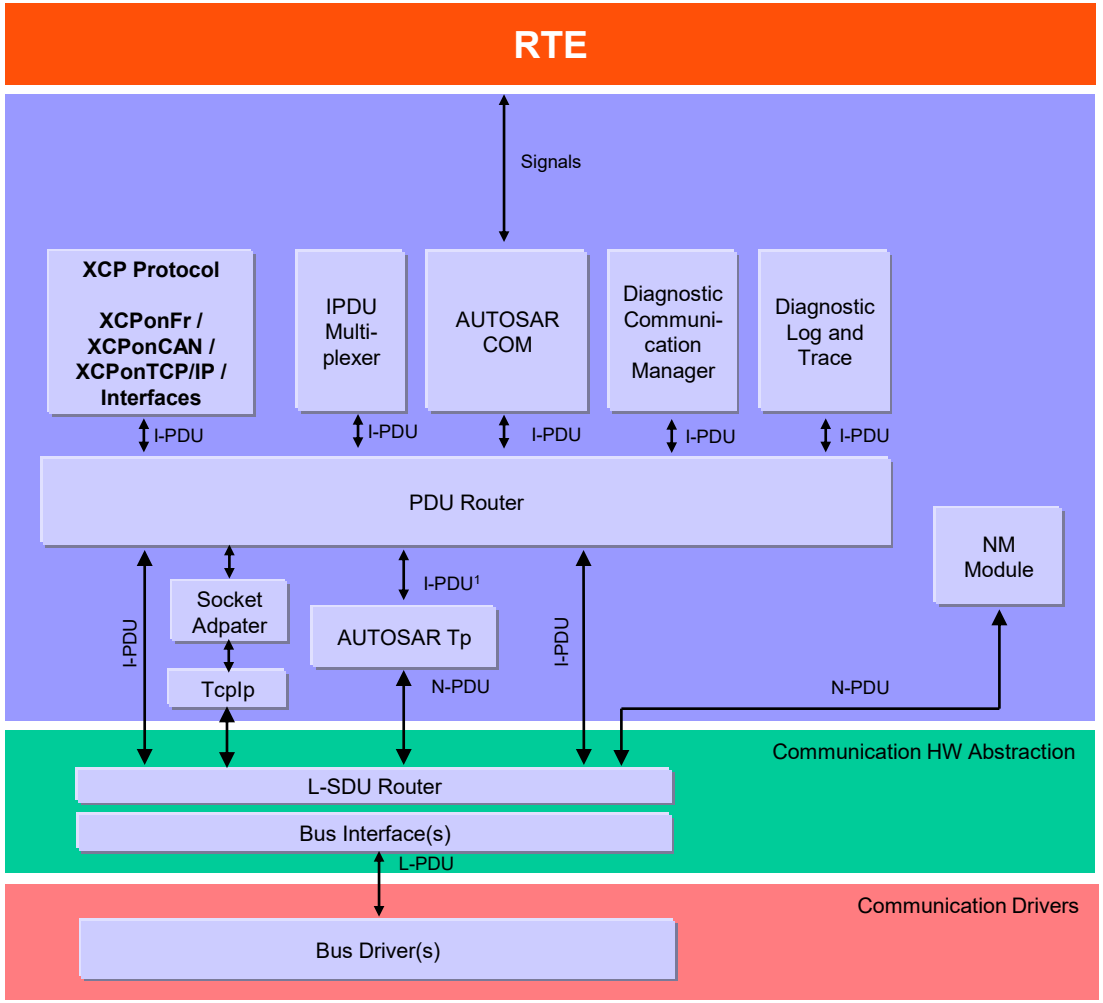
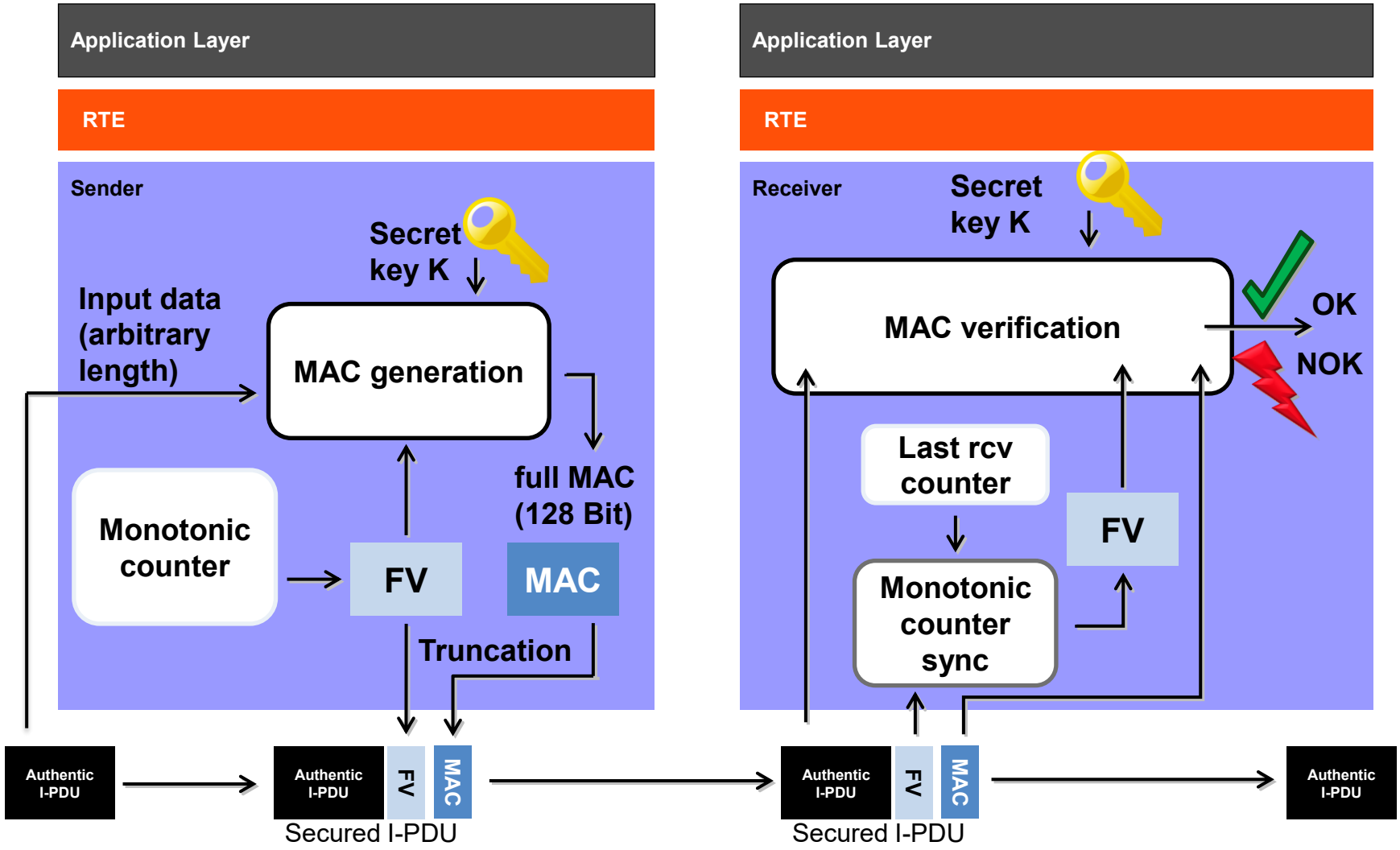


Table of contents

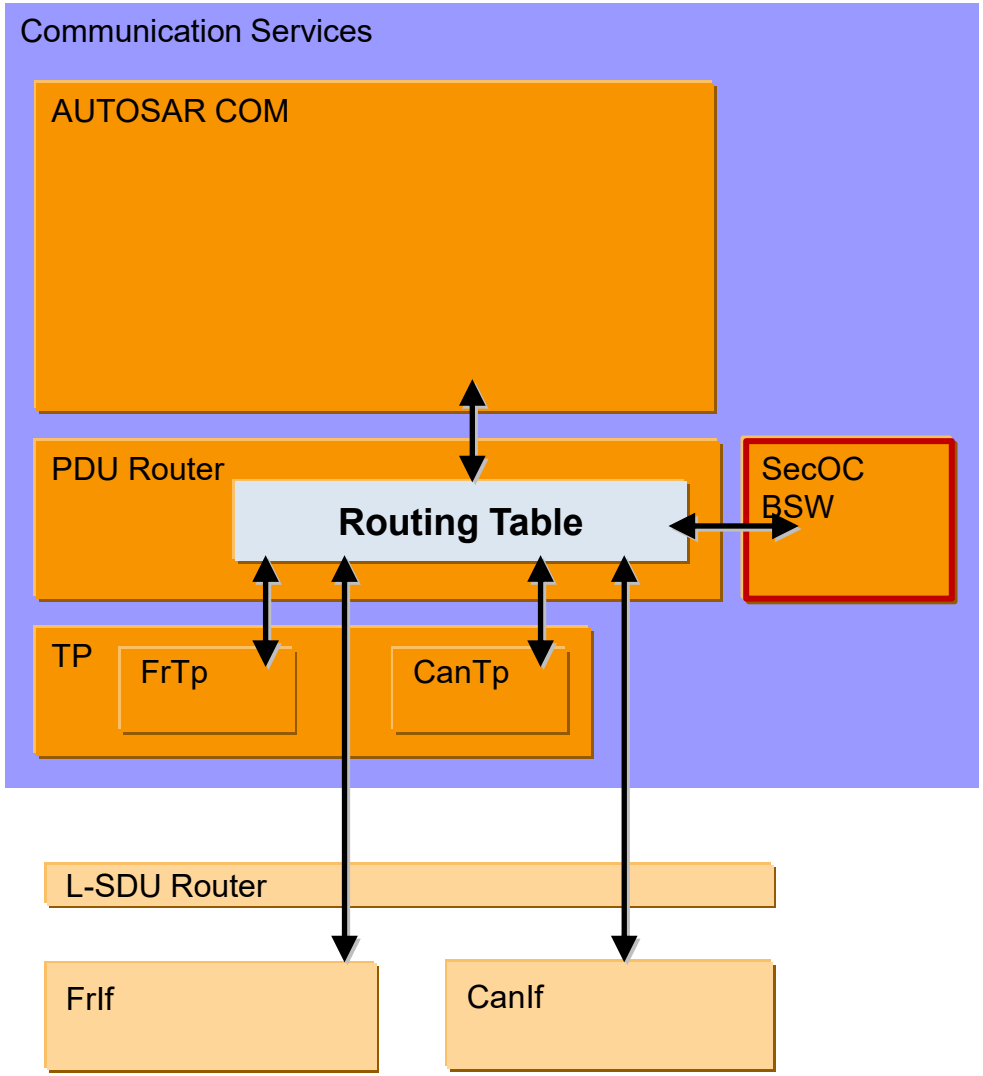
1. Architecture
2. Configuration
3. **Integration and Runtime Aspects**
    1. Mapping of Runnables
    2. Partitioning
    3. Scheduling
    4. Mode Management
    5. Error Handling, Reporting and Diagnostic
    6. Measurement and Calibration
    7. **Security**
    8. Energy Management
    9. Global Time Synchronization

Integration and Runtime Aspects – Secure Onboard Communication
Overview - Message Authentication and Freshness Verification



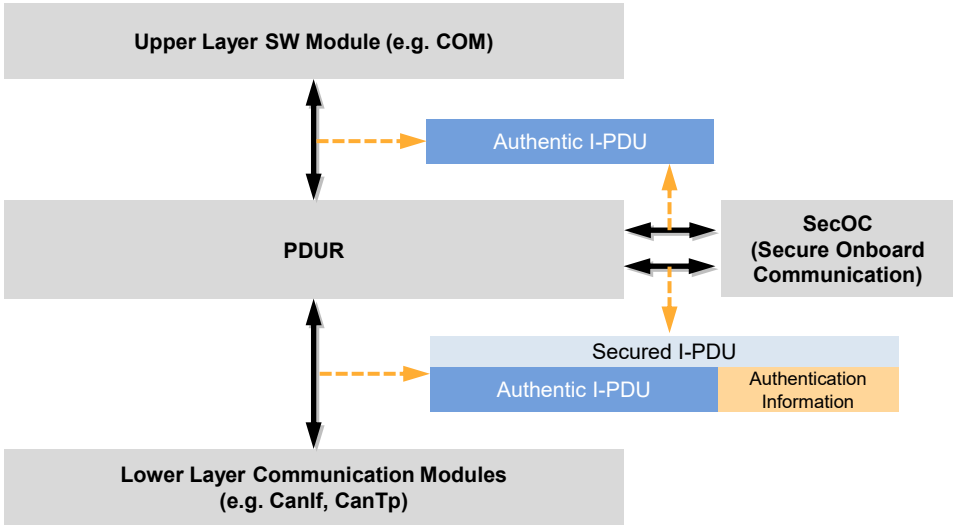
MAC: Message Authentication Code
FV: Freshness Counter Value

Integration and Runtime Aspects – Secure Onboard Communication
Integration as communication service

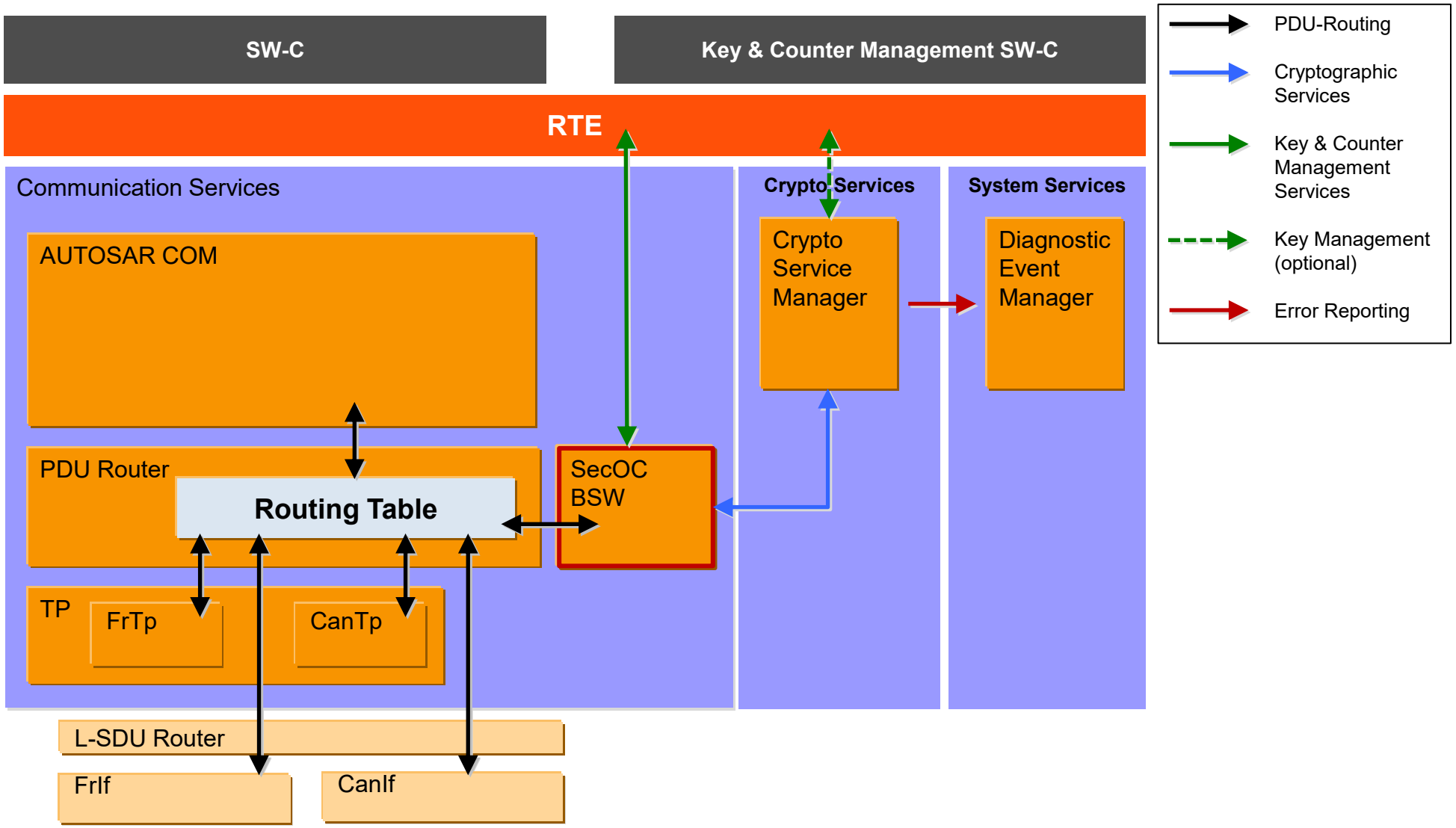


SecOC BSW:

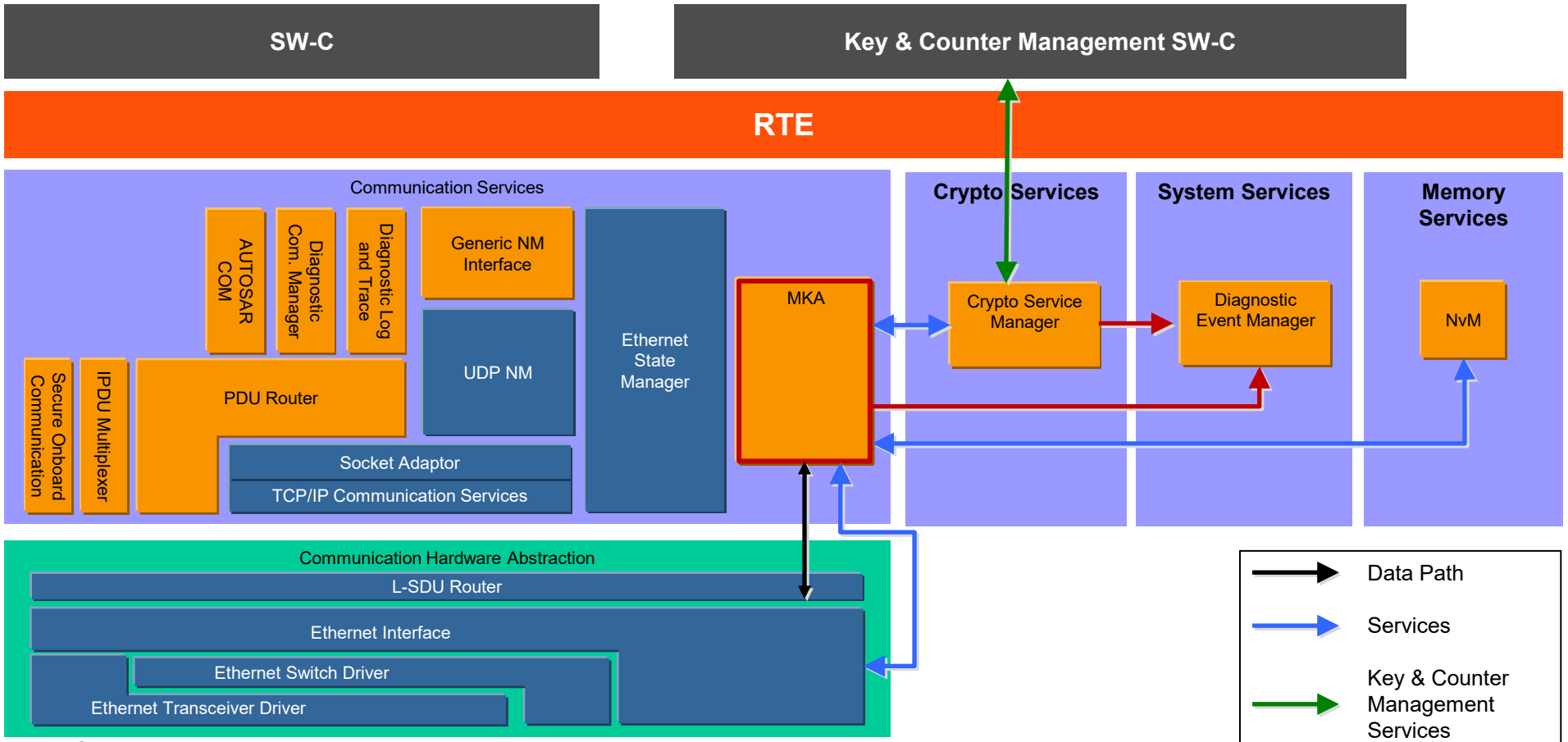
- adds/verifies authentication information (for/from lower layer)
- realizes interface of upper and lower layer modules
- is addressed by PduR routing configuration
- maintains buffers to store and modify secured I-PDUs



Integration and Runtime Aspects – Secure Onboard Communication
Integration with other services



Integration and Runtime Aspects – MACsec Key Agreement
Integration with other services



MACsec Key Agreement:

- Configures the MACsec entities to enable MACsec protected traffic
- Generates and processes MKPDUs
- Uses Crypto Services to generate and validate ICVs of MKPDUs

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. Measurement and Calibration
    7. Security
    8. **Energy Management**
    9. Global Time Synchronization

Energy Management
Introduction

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.

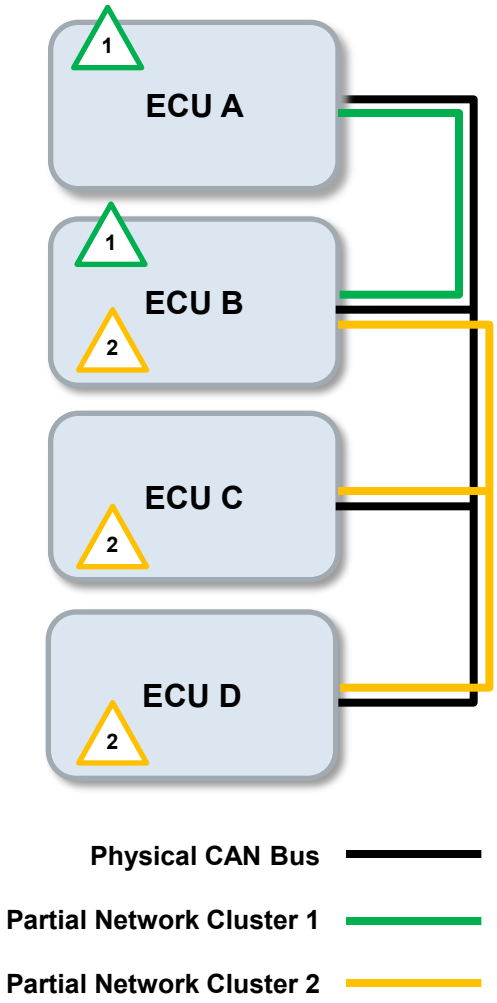
Partial Networking

- 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.

ECU Degradation

- Allows to switch of peripherals.

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.
- ECUs “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.

Energy Management – Partial Networking
Conceptual terms

- 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.

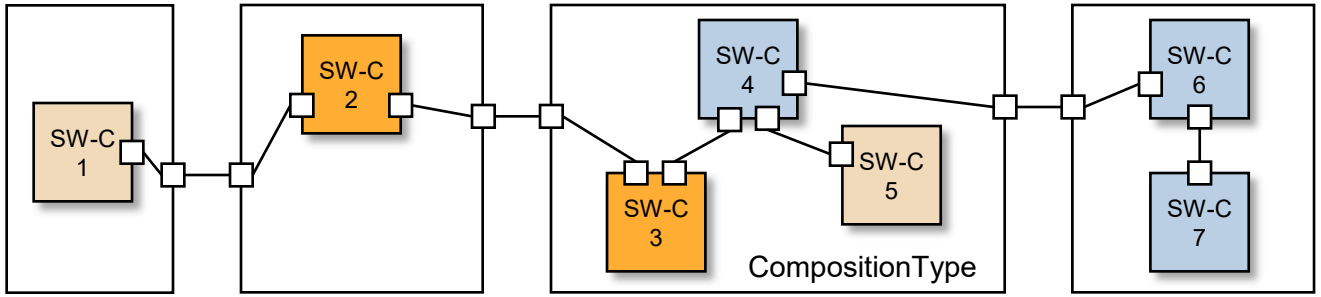
- **Partial Network Cluster (PNC):** is a group of system signals necessary to support one or more vehicle functions that are distributed across multiple ECUs in the vehicle network.
  This represents the system view of mapping a group of buses to one ore more VFCs.

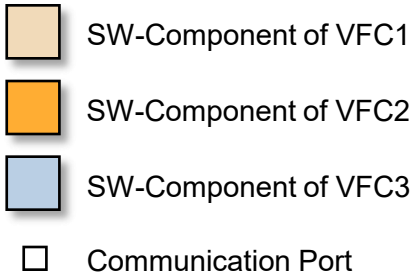
Energy Management – Partial Networking
Restrictions

- Partial Networking (PN) is currently supported on CAN and FlexRay buses.
- LIN and CAN slave buses (i.e. CAN buses without network management) can be activated* using PN but no wake-up or communication of NM messages (including a PNC Request Bit Vector) are supported on those buses
- To wake-up a PN ECU, a special transceiver HW is required as specified in ISO 11898-5.
  - The standard wake-up without special transceiver HW known from previous AUTOSAR releases is still supported.
- A VFC can be mapped to any number of PNCs (including zero)
  - The concept of PN considers a VFC with only ECU-internal communication by mapping it to the internal channel type in ComM as there is no bus communication and no physical PNC
- Restrictions for CAN
  - J1939 and PN exclude each other, due to address claiming and J1939 start-up behaviour
  - J1939 need to register first their address in the network before they are allowed to start communication after a wake-up.
  - A J1939 bus not using address claiming can however be activated using PN as a CAN slave bus as described above
- Restrictions on FlexRay
  - FlexRay is only supported for requesting and releasing PNCs.
  - FlexRay nodes cannot be shut down since there is no HW available which supports PN.

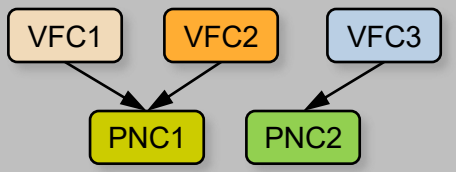
* All nodes connected to the slave buses are always activated. It is not possible only to activate a subset of the nodes.

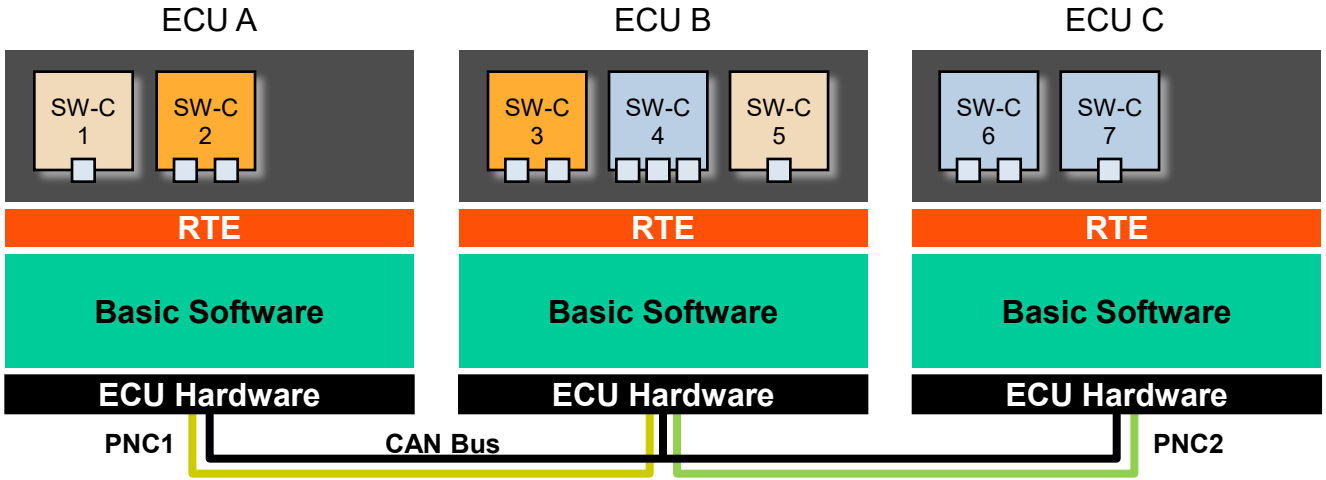
Energy Management – Partial Networking
Mapping of Virtual Function Cluster to Partial Network Cluster



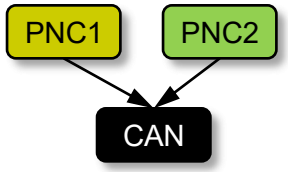


Mapping of VFC on PNC

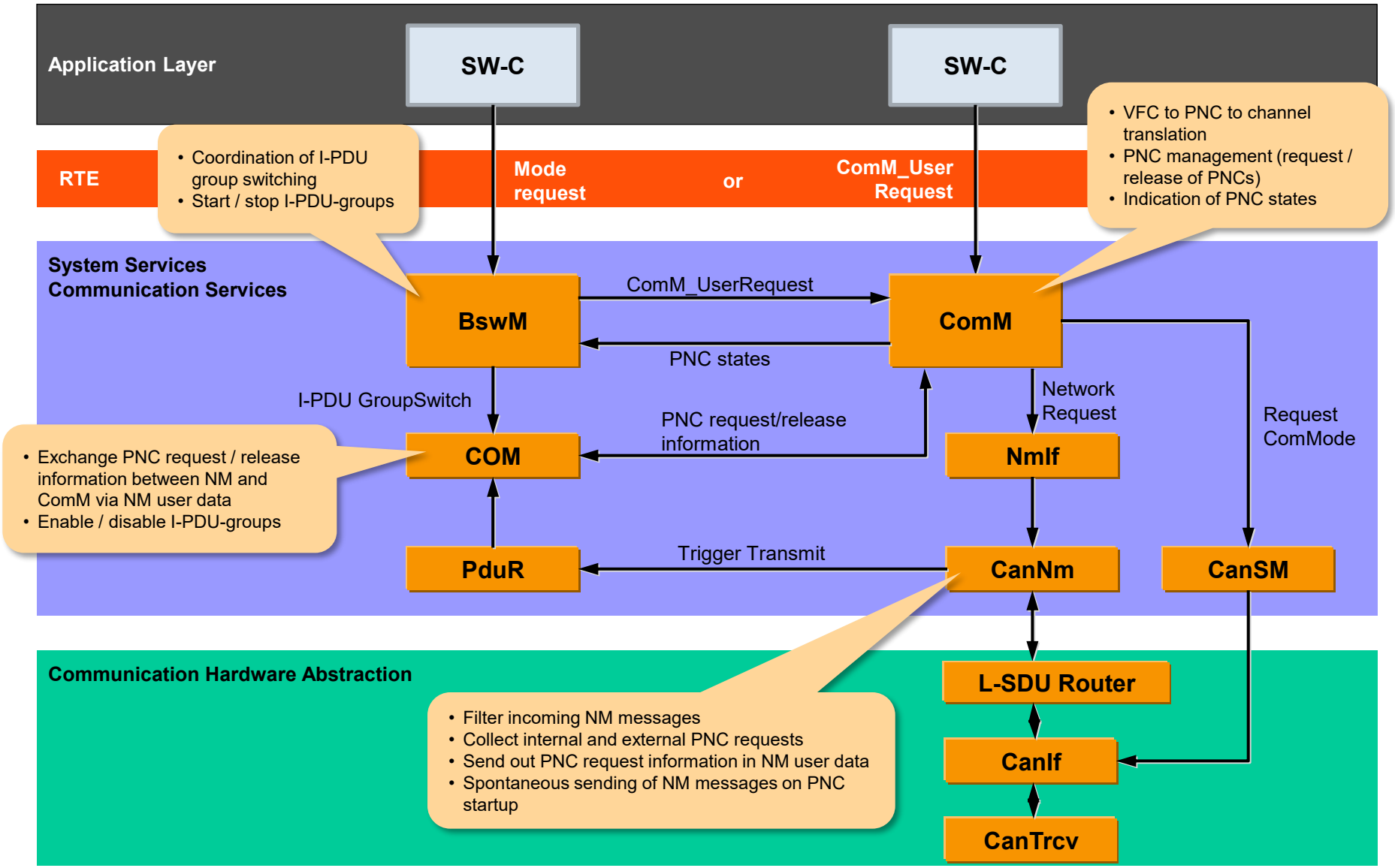




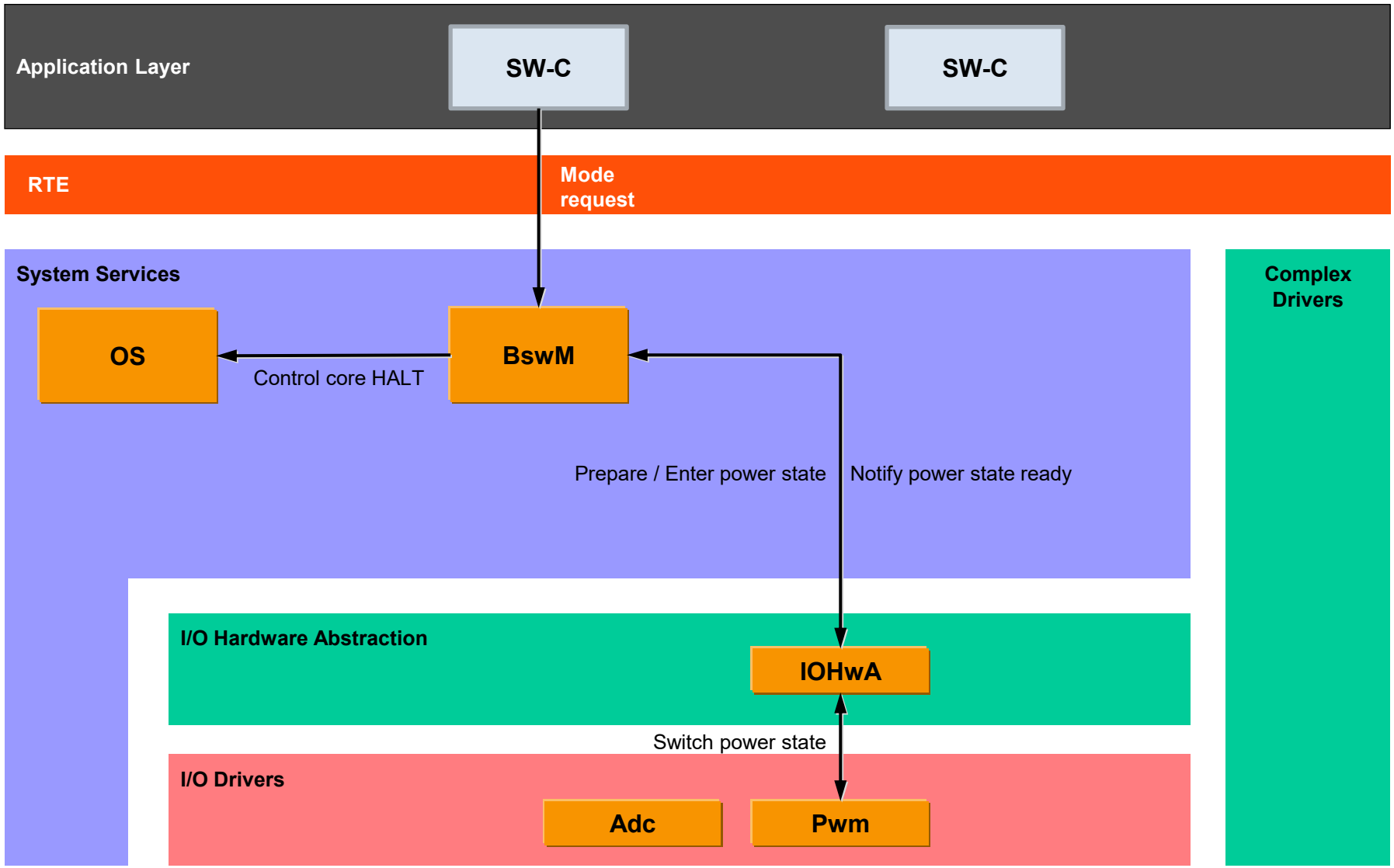
- Here both Partial Networks map to one CAN bus.
- One Partial Network can also span more than one bus.



Energy Management – Partial Networking
Involved modules – Solution for CAN



Energy Management – ECU Degradation
Involved modules – Solution for I/O Drivers



Energy Management – ECU Degradation
Restrictions

- ECU Degradation is currently supported only on MCAL drivers Pwm and Adc.
- Core HALT and ECU sleep are considered mutually exclusive modes.
- Clock modifications as a means of reducing power consumption are not in the scope of the concept (but still remain available as specific MCU driver configurations).

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. Measurement and Calibration
    7. Security
    8. Energy Management
    9. **Global Time Synchronization**

Integration and Runtime Aspects – Global Time Synchronization

Global Time Synchronization provides synchronized time base(s) over multiple in-vehicle networks.

StbM provides the following **features**:

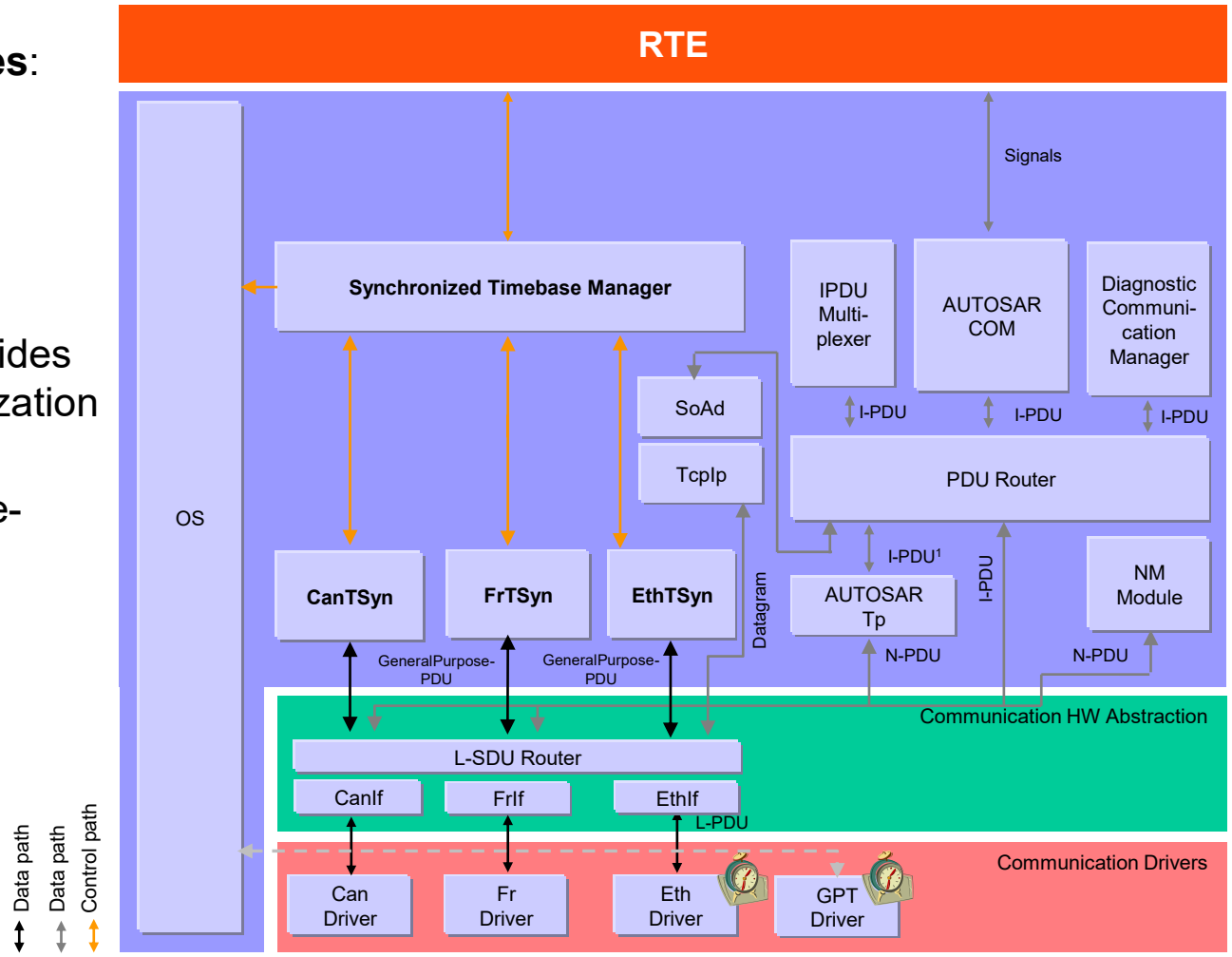
- Time provision
- Time base status
- Time gateway

CanTSyn / FrTSyn / EthTSyn provides the network-specific time synchronization protocol.

EthTSyn provides additionally a rate-correction and latency calculation.

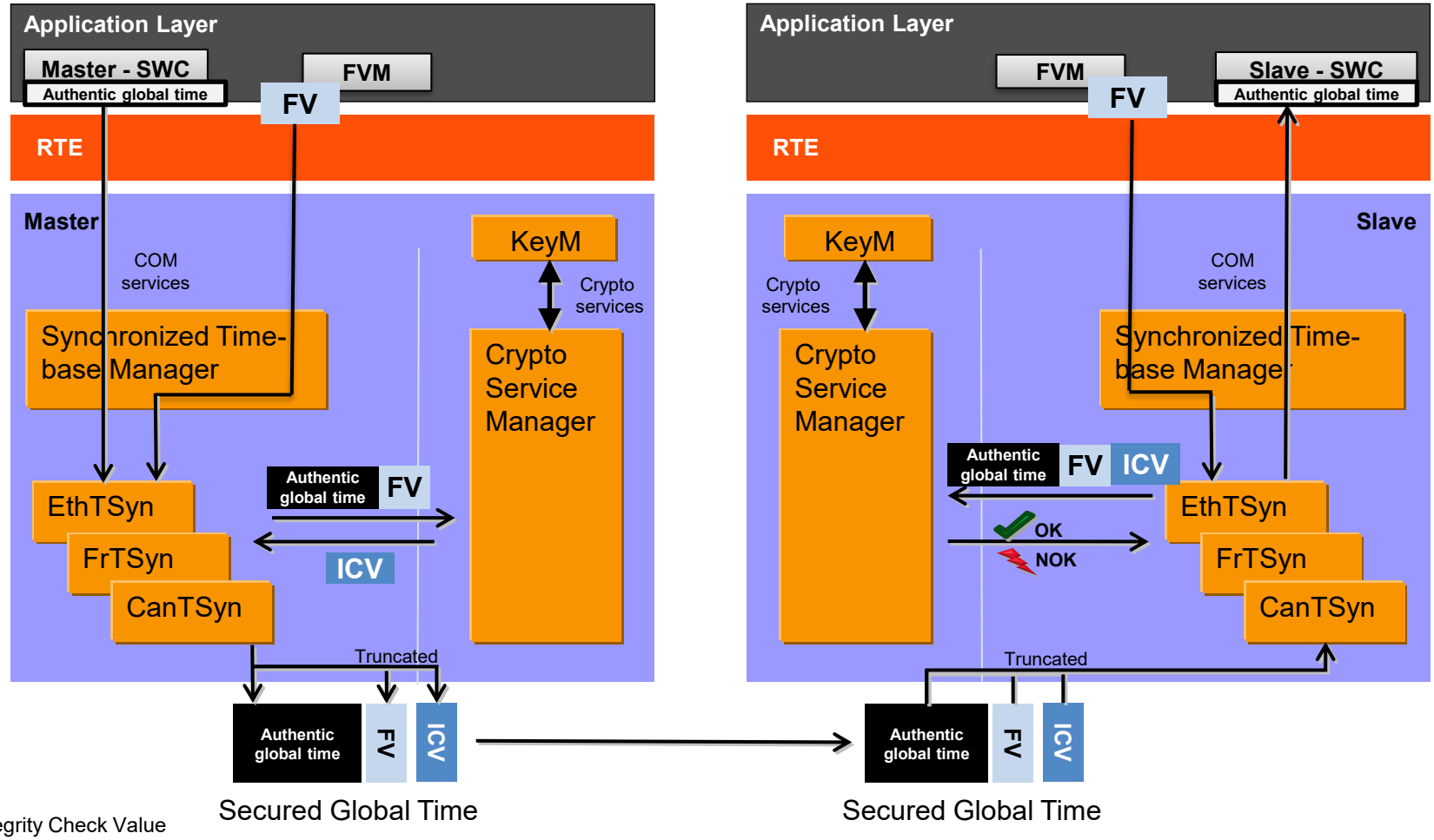
Use-case examples:

- Sensor data fusion
- Cross-ECU logging



Integration and Runtime Aspects – Secure Global Time Synchronization

Secure Global Time Synchronization ensures integrity and authenticity of synchronized time base(s) over in-vehicle networks.



ICV: Integrity Check Value
FV: Freshness Value
FVM: Freshness Value Manager