Layered Software Architecture

- AUTOSAR Confidential -

Document Information

Document Title	Layered Software Architecture
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	053
Document Classification	Auxiliary
Document Version	2.4.0
Document Status	Final
Part of Release	3.2
Revision	3

Document Information

Document Change History			
Date	**Version**	**Changed by**	**Change Description**
28.02.2014	2.4.0	AUTOSAR Release Management	• Editorial changes
02.04.2012	2.3.0	AUTOSAR Administration	• Moved error handling into new chapter „integration and runtime aspects“ • Clarified generated DEM symbolic names according ecuc_sws_2108 • Removed blocks for Time- and Synchronization Services • Added BSW Mode Manager • Retrofitting of CDD concept • Added a new chapter for Partial Networking concept • Fixed typos

Document Information

Document Change History			
Date	**Version**	**Changed by**	**Change Description**
18.03.2011	2.2.2	AUTOSAR Administration	• Legal disclaimer revised
23.06.2008	2.2.1	AUTOSAR Administration	• Legal disclaimer revised
15.11.2007	2.2.0	AUTOSAR Administration	• Updates based on new wakeup/startup concepts • Detailed explanation for post-build time configuration • "Slimming" of LIN stack description • ICC2 figure • Document meta information extended • Small layout adaptations made
06.02.2007	2.1.0	AUTOSAR Administration	• ICC clustering added. • Document contents harmonized • Legal disclaimer revised • Release Notes added • “Advice for users” revised • “Revision Information” added
21.03.2006	2.0.0	AUTOSAR Administration	Rework Of: • Error Handling • Scheduling Mechanisms More updates according to architectural decisions in R2.0
31.05.2005	1.0.1	AUTOSAR Administration	Correct version released
09.05.2005	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

Disclaimer

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

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

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.

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

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

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice for users

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

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

ID: 00 – Overview

Overview

Part 1 – Introduction, Scope and Limitations
Part 2 – Overview of Software Layers
Part 3 – Contents of Software Layers
Part 4 – Interfaces
- 4.1 General Rules
- 4.2 Interaction of Layers – Example "Memory"
- 4.3 Interaction of Layers – Example "Communication"
- 4.4 Interaction of Layers – Example "ECU State Manager"

Part 5 – Configuration
Part 6 – Scheduling
Part 7 – Implementation Conformance Classes
Part 8 – Integration and Runtime aspects
- 8.1 Error Handling and Reporting Concept
- 8.2 Partial Networking

Part 1 – Introduction, Scope and Limitations

- AUTOSAR Confidential -

Part 1 – Introduction, Scope and Limitations
ID: 01-01

Purpose of this document

The **Layered Software Architecture** maps the identified modules of the **Basic Software** Module List to software layers and shows their relationship.

This document does not contain requirements. It is a document summarizing architectural decisions and discussions of AUTOSAR. The examples given are not meant to be complete in all respects.

This document focuses on static views of a conceptual layered software architecture. This document does not specify a structural software architecture with detailed static and dynamic interface descriptions. This is included in the specifications of the basic software modules.

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

Inputs and requirements

This document has been generated based on following documents:

- Basic Software Module List
- Specification of Virtual Functional Bus
- Several views of automotive ECU software architectures

- AUTOSAR Confidential -

Part 1 – Introduction, Scope and Limitations
ID: 01-02

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

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

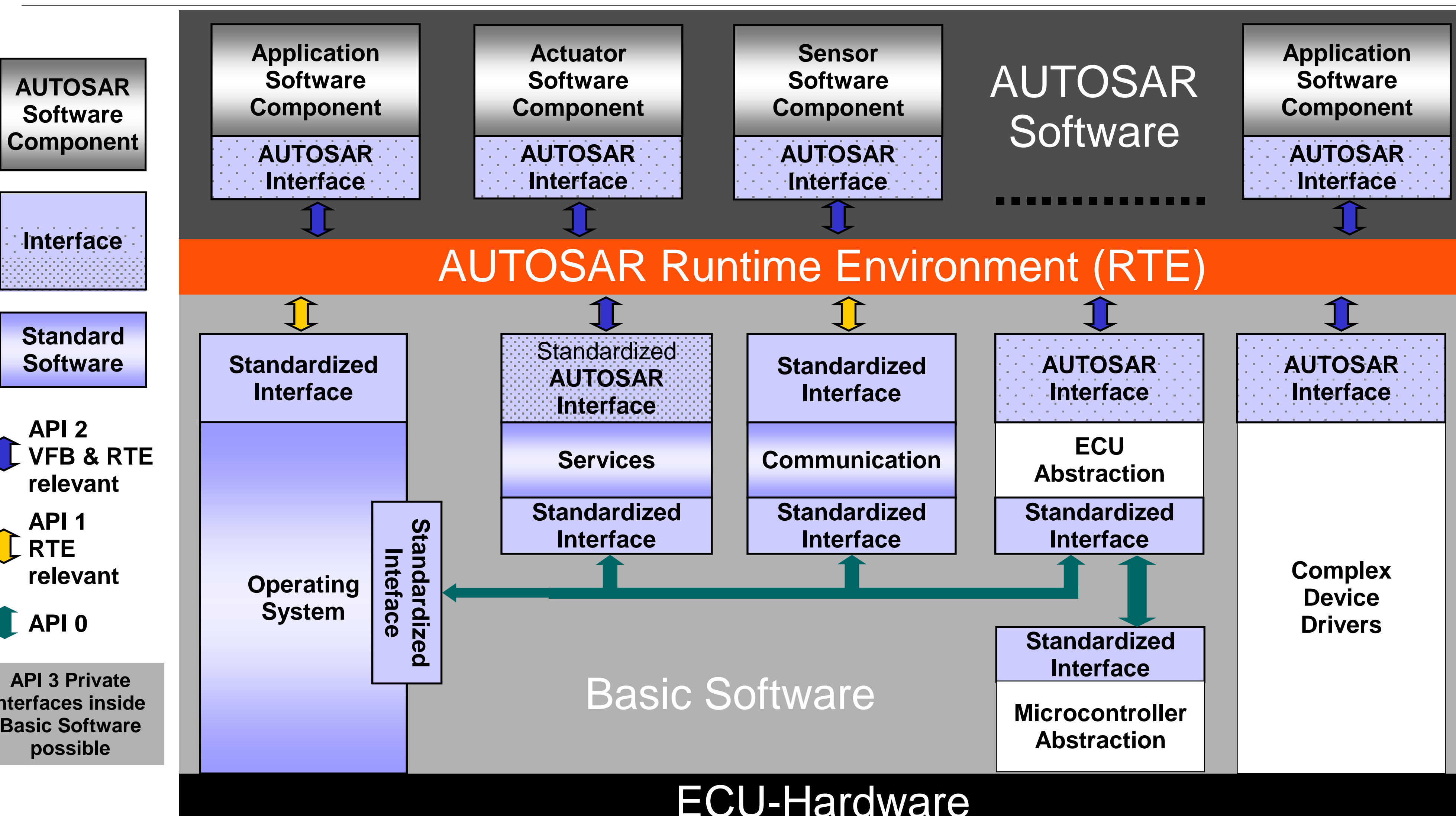
Extensibility:
- This SW Architecture is a generic approach. Modules can be added or existing ones can be extended in functionality, but their configuration has to be considered in the automatic Basic SW configuration process!
- Complex drivers can easily be added
- Further Layers cannot be added

- AUTOSAR Confidential -

ID: 02 – Layered Software Architecture

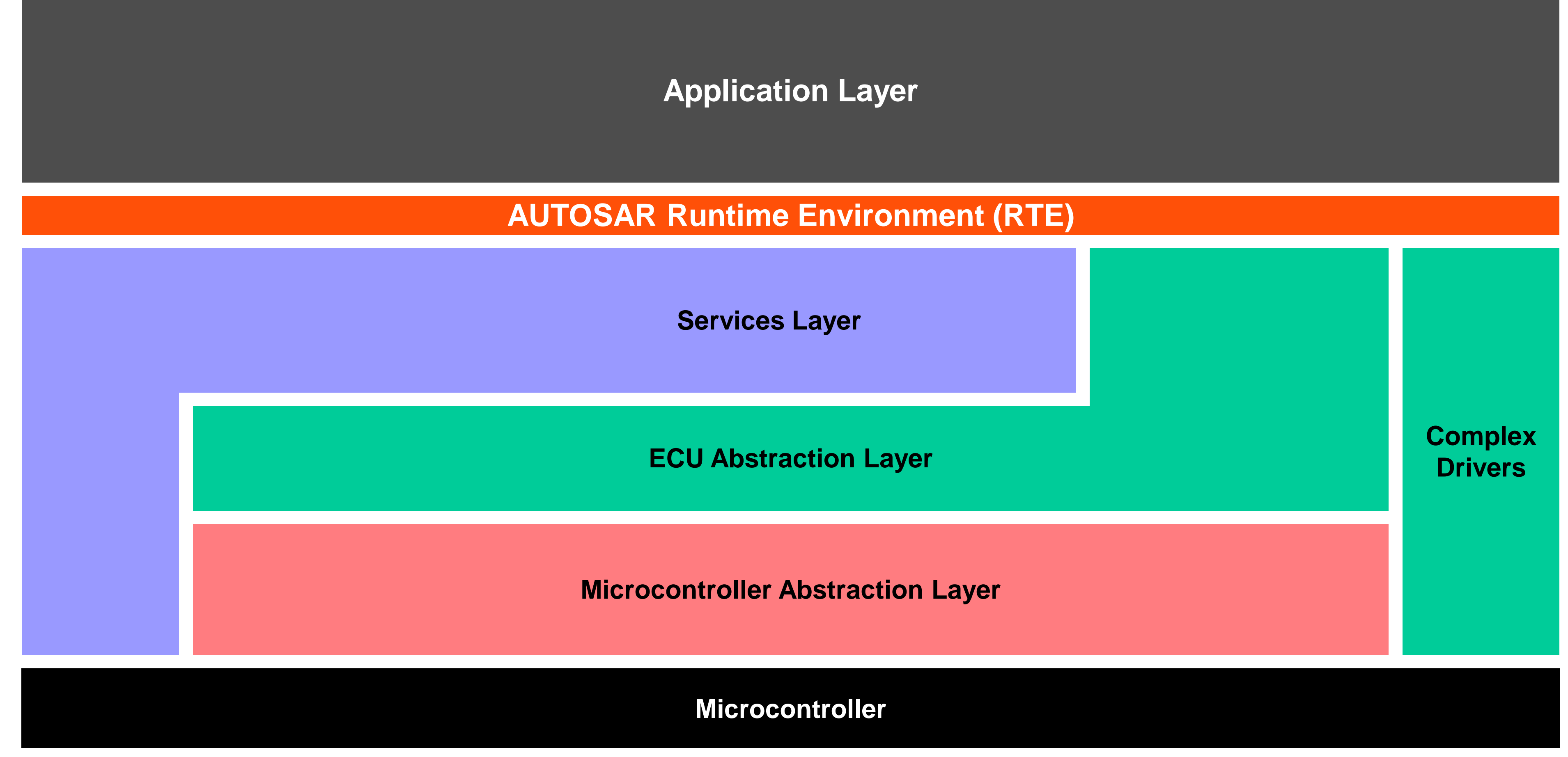
Part 2 – Overview of Software Layers

Part 2 – Overview of Software Layers
ID: 02-01 Simplified Component View

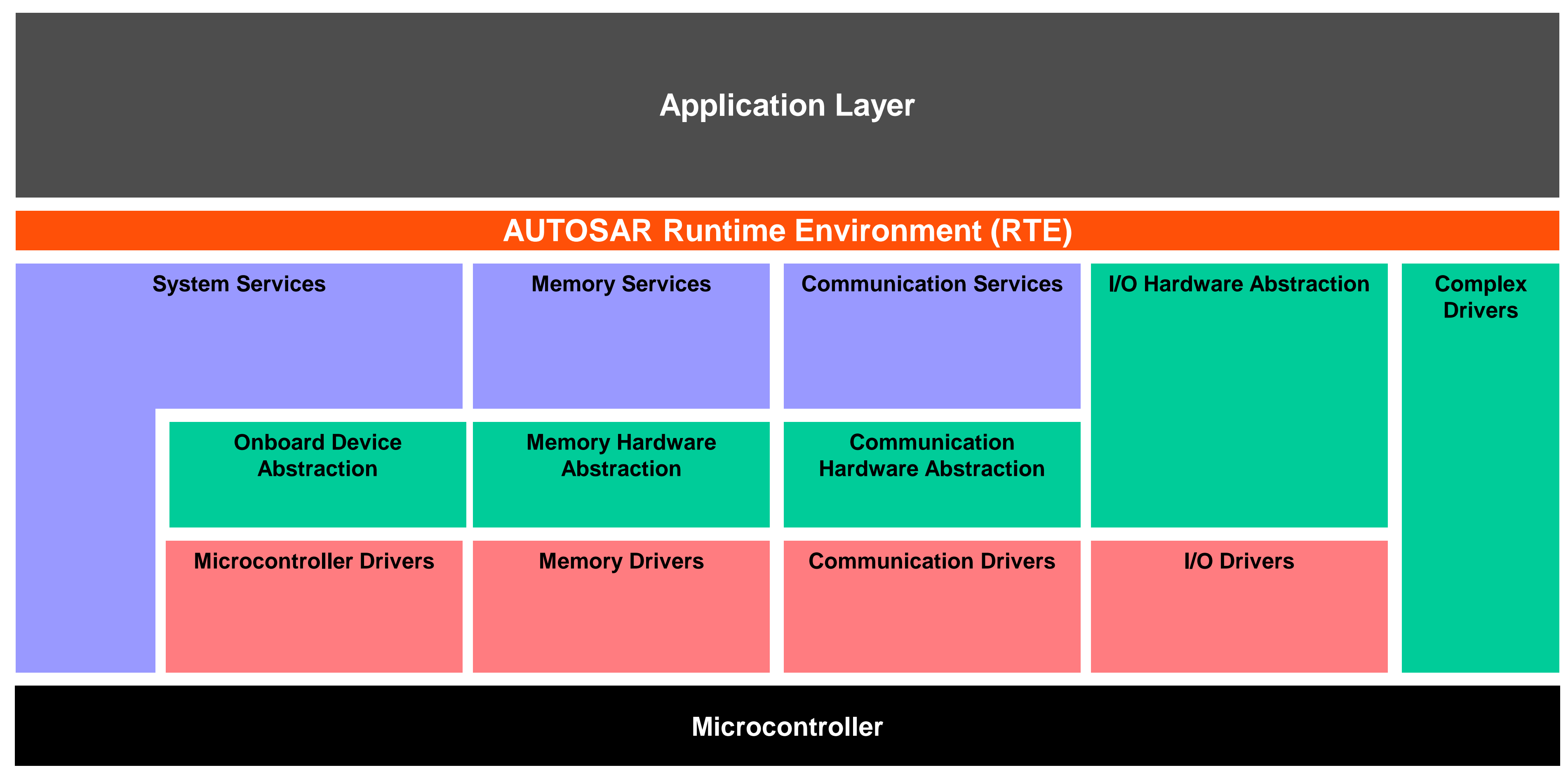


Note: This figure is incomplete with respect to the possible interactions between the layers. Please refer to slide ID 04-003 for additional details.

Part 2 – Overview of Software Layers
ID: 02-02 Layered View: Coarse



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

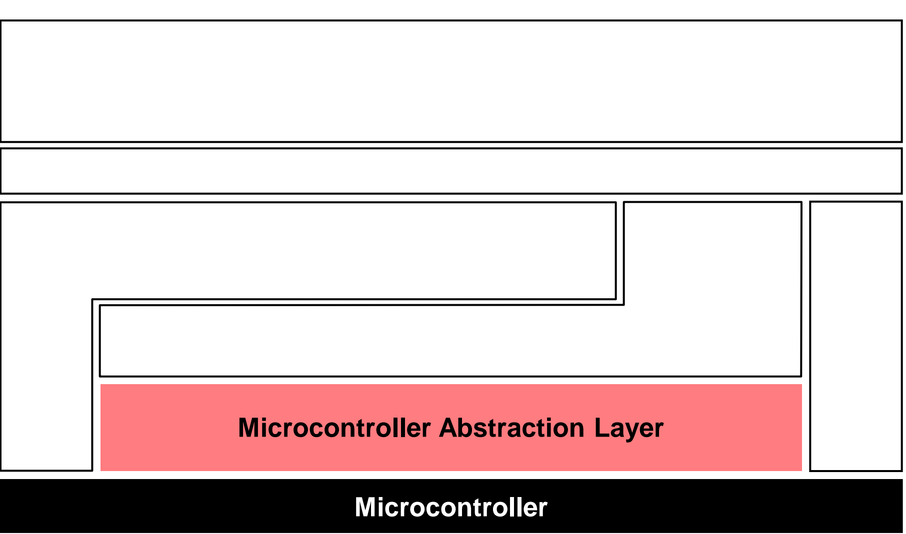


Part 2 – Overview of Software Layers
ID: 02-04 Introduction to Basic Software Layers (1)

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 internal peripherals and memory mapped µC external devices.

Task:
Make higher software layers independent of µC

Properties:
Implementation: µC dependent
Upper Interface: standardizable and µC independent

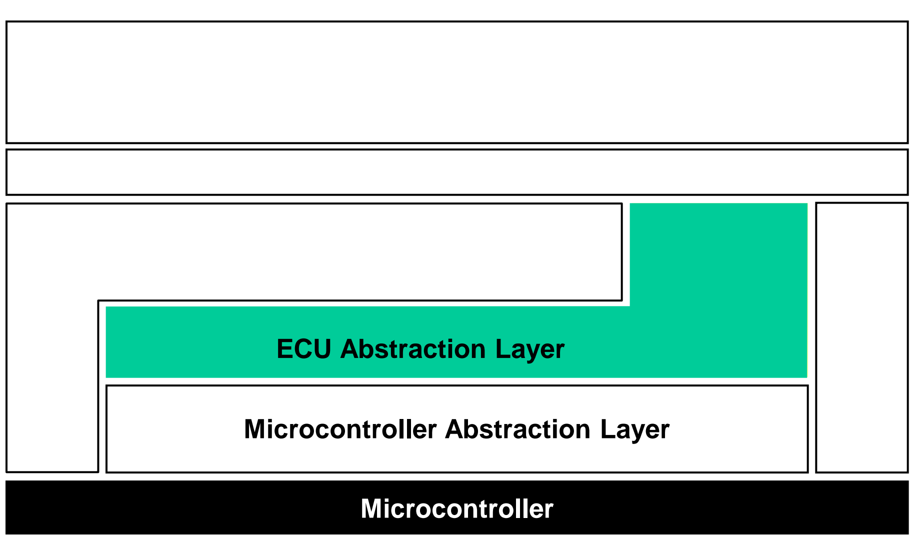


Part 2 – Overview of Software Layers
ID: 02-05 Introduction to Basic Software Layers (2)

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

Task:
Make higher software layers independent of ECU hardware layout

Properties:
Implementation: µC independent, ECU hardware dependent
Upper Interface: µC and ECU hardware independent, dependent on signal type



Part 2 – Overview of Software Layers
ID: 02-06 Introduction to Basic Software Layers (3)

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

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

Services Layer
ECU Abstraction Layer
Microcontroller Abstraction Layer
Microcontroller

- AUTOSAR Confidential -

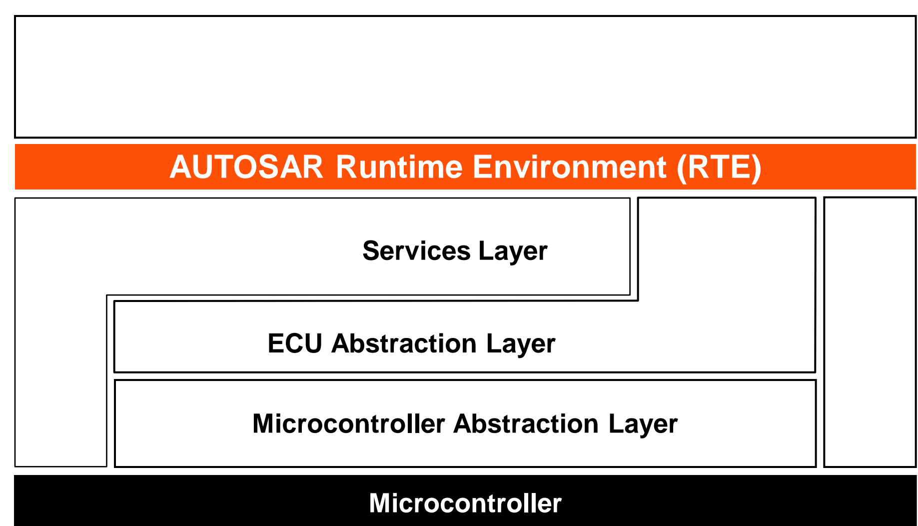
Part 2 – Overview of Software Layers
ID: 02-07 Introduction to Basic Software Layers (4)

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



Part 2 – Overview of Software Layers
ID: 02-08 Introduction to Basic Software Layers (5)

The **Basic Software** can be subdivided into the following types of services:

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

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

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

- **System**
  Provision of standardisable (operating system, timers, error memory) and ECU specific (ECU state management, watchdog manager) services and library functions

- AUTOSAR Confidential -

Part 2 – Overview of Software Layers
ID: 02-09 Introduction to Basic Software Module Types (1)

Driver
A **driver** contains the functionality to control and access an internal or an external device.

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

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

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

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

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

Exception:
The drivers for memory mapped external devices (e.g. external flash memory) may access the microcontroller directly. Those external drivers are located in the Microcontroller Abstraction Layer because they are microcontroller dependent.

- AUTOSAR Confidential -

Part 2 – Overview of Software Layers
ID: 02-10 Introduction to Basic Software Module Types (2)

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

The interface does not change the content of the data.

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

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

- AUTOSAR Confidential -

Part 2 – Overview of Software Layers
ID: 02-11 Introduction to Basic Software Module Types (3)

Manager

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

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

In general, managers are located in the **Services Layer**

Example: The NVRAM manager manages the concurrent access to internal and/or external memory devices like flash and EEPROM memory. It also performs management of RAM mirrors, redundant, distributed and reliable data storage, data checking, provision of default values etc. For details refer to the AUTOSAR requirements documents.

- AUTOSAR Confidential -

ID: 03 – Layered Software Architecture

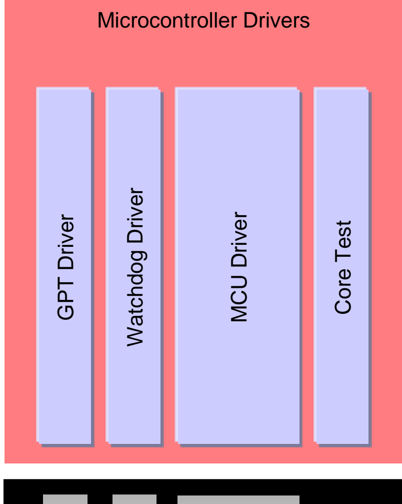
Part 3 – Contents of Software Layers

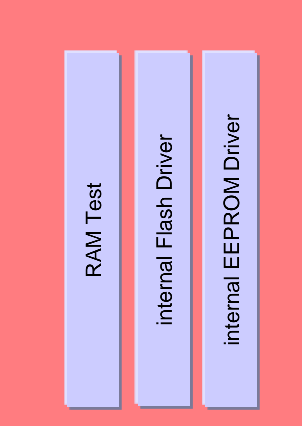
- AUTOSAR Confidential -

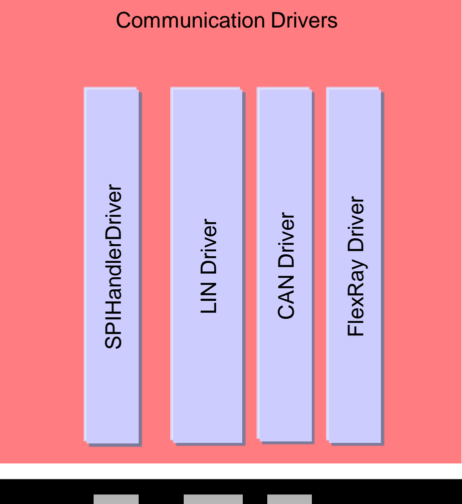
Part 3 – Contents of Software Layers
ID: 03-01 Scope: Microcontroller Abstraction Layer

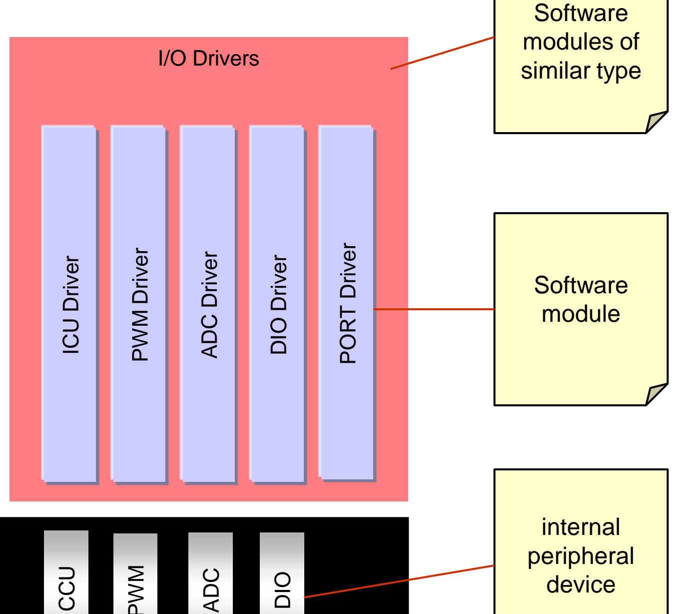
The **µC Abstraction Layer** consists of the following module groups:

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









Part 3 – Contents of Software Layers
ID: 03-02 Scope: Complex Drivers

A **Complex Driver** 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.

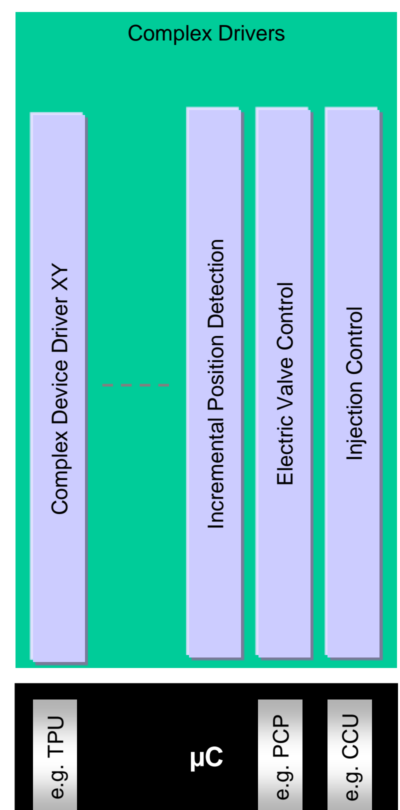
- Injection control
- Electric valve control
- Incremental position detection

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

Properties:
Implementation: highly µC, ECU and application dependent
Upper Interface: specified and implemented according to AUTOSAR (AUTOSAR interface)

Micro-controller Drivers
Memory Drivers
Communi-cation Drivers
I/O Drivers
Complex Drivers
Microcontroller (µC)

Example:

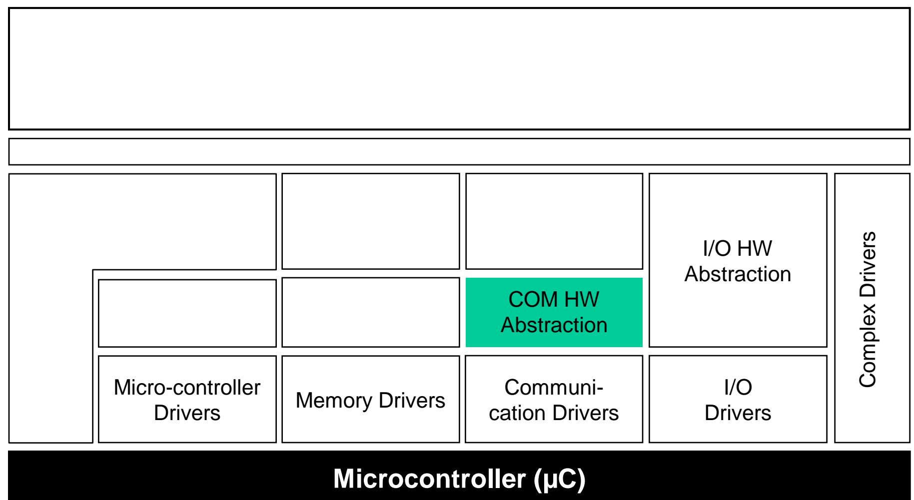


Part 3 – Contents of Software Layers
ID: 03-04 Scope: Communication Hardware Abstraction

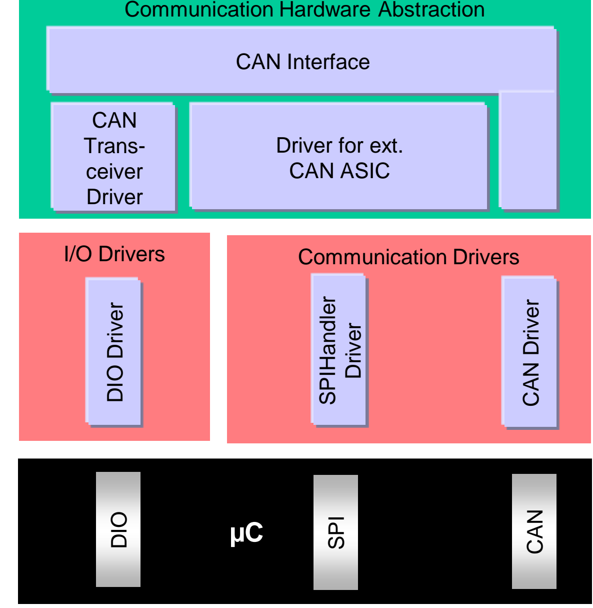
The **Communication Hardware Abstraction** is a group of modules which abstracts from the ***location*** of communication controllers and the ***ECU hardware layout***. For all communication systems a ***specific*** Communication Hardware Abstraction is required (e.g. for LIN, CAN, FlexRay).
Example: An ECU has a microcontroller with 2 internal CAN channels and an additional on-board ASIC with 4 CAN controllers. The CAN-ASIC is connected to the microcontroller via SPI.
The communication drivers are accessed via bus specific interfaces (e.g. CAN Interface).

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

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



Example:

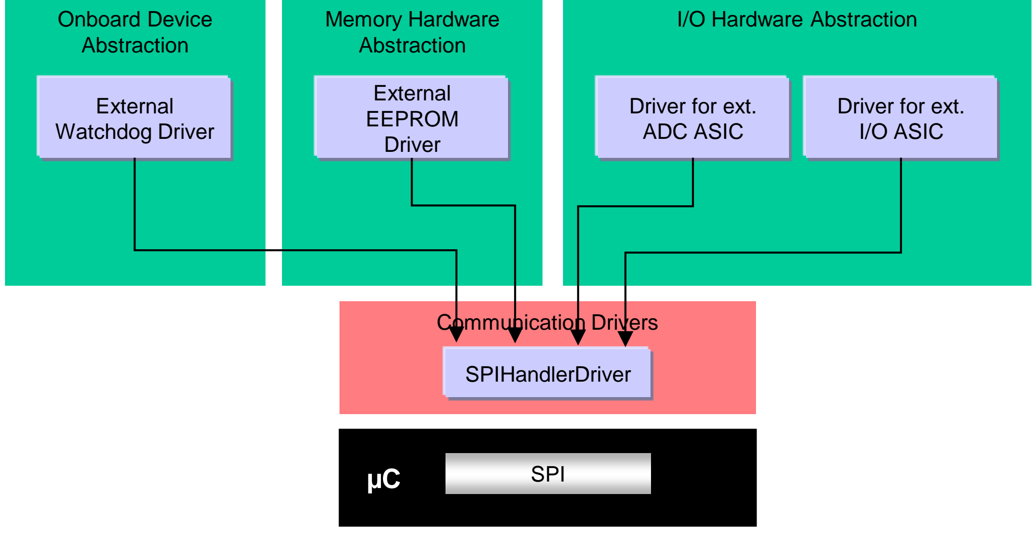


Part 3 – Contents of Software Layers
ID: 03-16 Scope: SPIHandlerDriver

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:



- AUTOSAR Confidential -

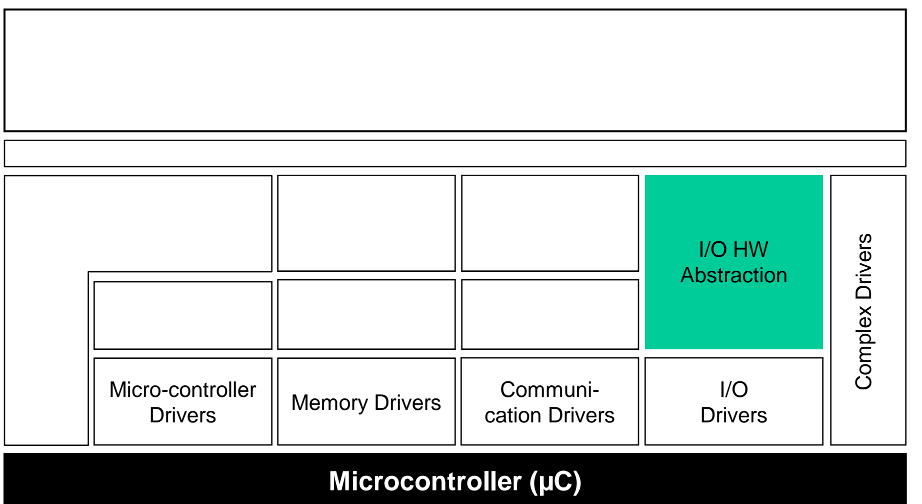
Part 3 – Contents of Software Layers
ID: 03-03 Scope: I/O Hardware Abstraction

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!

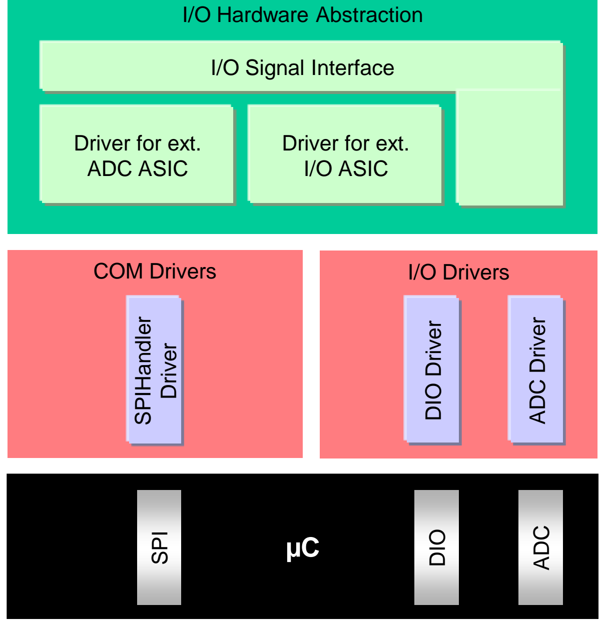
The different I/O devices are accessed via an I/O signal interface.

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

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



Example:



- AUTOSAR Confidential -

Part 3 – Contents of Software Layers
ID: 03-05 Scope: Memory Hardware Abstraction

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

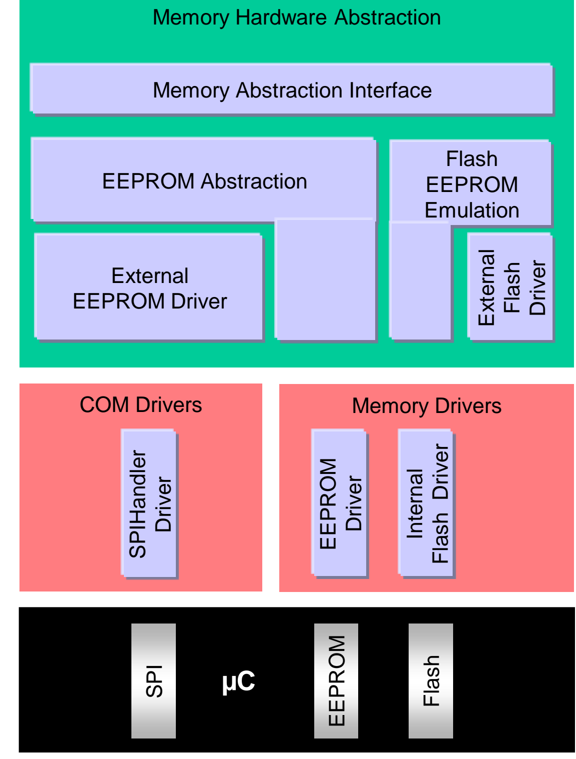
The memory drivers are accessed via memory specific abstraction/emulation modules (e.g. EEPROM Abstraction).
By emulating an EEPROM abstraction on top of Flash hardware units a common access via Memory Abstraction Interface to both types of hardware is enabled.

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

I/O HW Abstraction
Memory HW Abstraction
COM HW Abstraction
Complex Drivers
Micro-controller Drivers
Memory Drivers
Communi-cation Drivers
I/O Drivers
Microcontroller (µC)

Example:



Part 3 – Contents of Software Layers
ID: 03-06 Scope: Onboard Device Abstraction

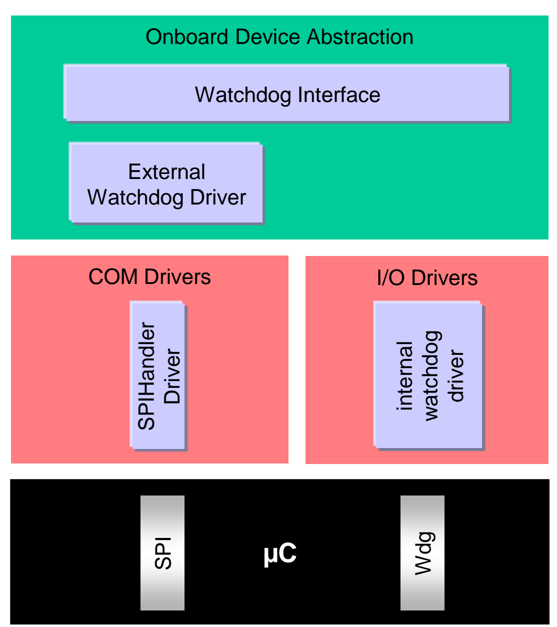
The **Onboard Device Abstraction** contains drivers for ECU onboard devices which cannot be seen as sensors or actuators like internal or external watchdogs. Those drivers access the ECU onboard devices via the µC Abstraction Layer.

Task:
Abstract from ECU specific onboard devices.

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

Onboard Dev. Abstraction
Memory HW Abstraction
COM HW Abstraction
I/O HW Abstraction
Complex Drivers
Micro-controller Drivers
Memory Drivers
Communi-cation Drivers
I/O Drivers
Microcontroller (µC)

Example:



- AUTOSAR Confidential -

Part 3 – Contents of Software Layers
ID: 03-07 Scope: Communication Services – General

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

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

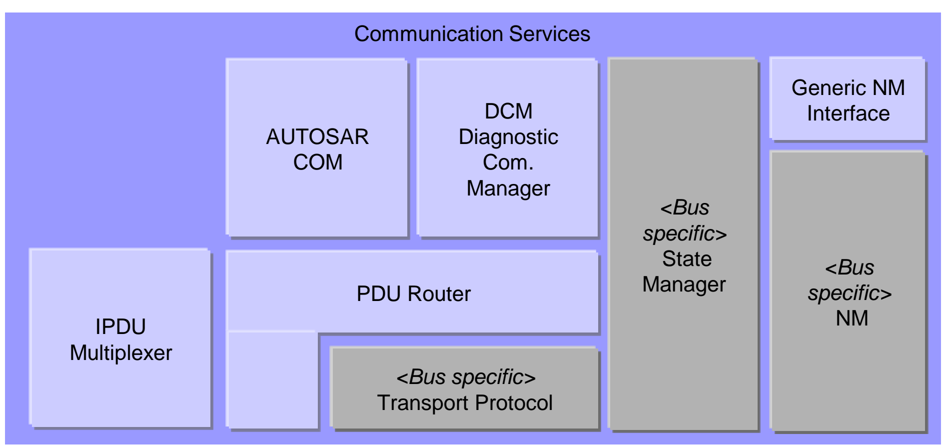
Properties:
Implementation: µC and ECU HW independent, partly dependent on bus type
Upper Interface: µC, ECU hardware and bus type independent

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

Application Layer

Communi-cation Services
Onboard Dev. Abstraction
Memory HW Abstraction
COM HW Abstraction
I/O HW Abstraction
Complex Drivers
Micro-controller Drivers
Memory Drivers
Communi-cation Drivers
I/O Drivers

Microcontroller (µC)



Color code: Bus specific modules are marked gray.

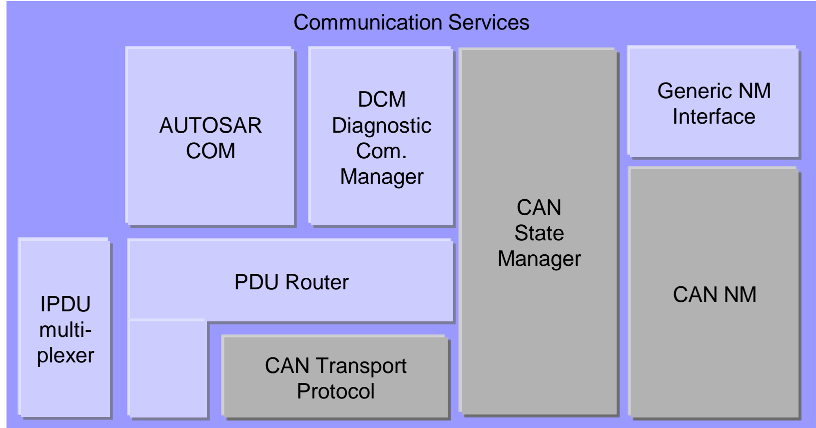
- AUTOSAR Confidential -

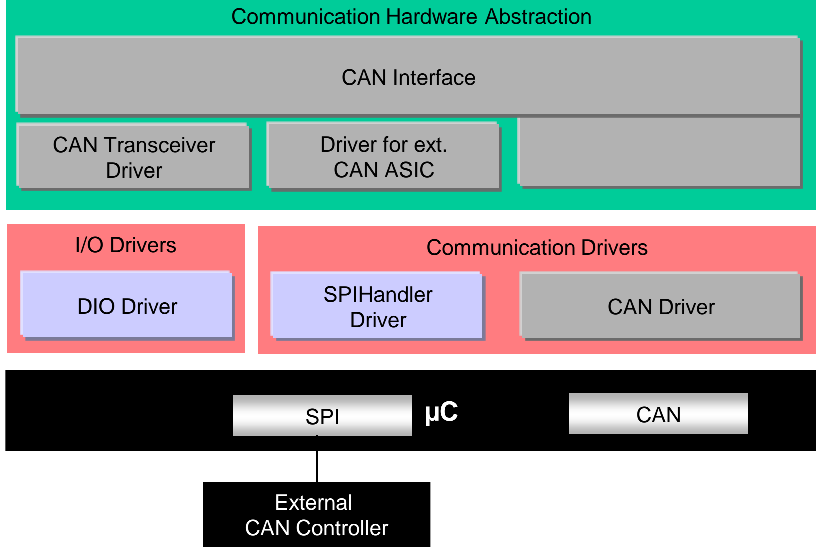
Part 3 – Contents of Software Layers
ID: 03-08 Scope: Communication Stack – CAN

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

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

Properties:
- Implementation: µC and ECU HW independent, partly dependent on CAN.
- AUTOSAR COM, 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 GW 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.
- CAN Generic NM is specific for CAN networks and will be instantiated per CAN vehicle network system. CAN Generic NM interface with CAN via underlying network adapter (CAN NM).
- The communication system specific Can State Manager handles the communication system dependent Start-up and Shutdown features. Furthermore it controls the different options of COM to send PDUs and to monitor signal timeouts.
- A signal gateway is part of AUTOSAR COM to route signals.
- PDU based Gateway is part of PDU router.
- IPDU multiplexing provides the possibility to add information to enable the multiplexing of I-PDUs (different contents but same IDs).
- Upper Interface: µC, ECU hardware and network type independent (goal)
- For refinement of GW architecture please refer to slide 04-050.





- AUTOSAR Confidential -

Part 3 – Contents of Software Layers
ID: 03-15 Scope: Communication Stack – LIN

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

Task:

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

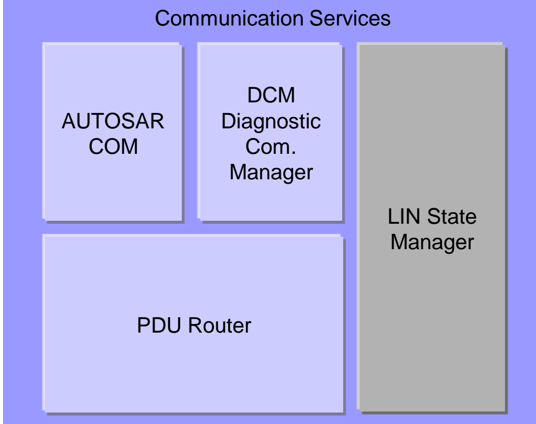
Properties:

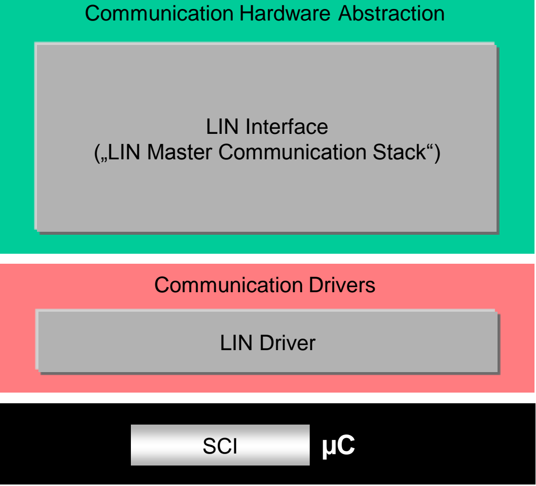
The **LIN Communication Services** contain:

- A LIN 2.0 compliant communication stack with
  - Schedule table manager for transmitting LIN frames and to handle requests to switch to other schedule tables.
  - Transport protocol, used for diagnostics
  - A WakeUp and Sleep Interface
- An underlying LIN Driver:
  - implementing the LIN protocol and adaptation the specific hardware
  - Supporting both simple UART and complex frame based LIN hardware

Note: Integration of LIN into AUTOSAR:

- The scheduler manager and its interfaces are used to decide the point of time to send a LIN frame.
- Lin Interface controls the WakeUp/Sleep API and allows the slaves to keep the bus awake (decentralized approach).
- The PDU router accesses the LIN Interface on PDU-Level, not on signal level.
- 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. after sending the LIN frame header).





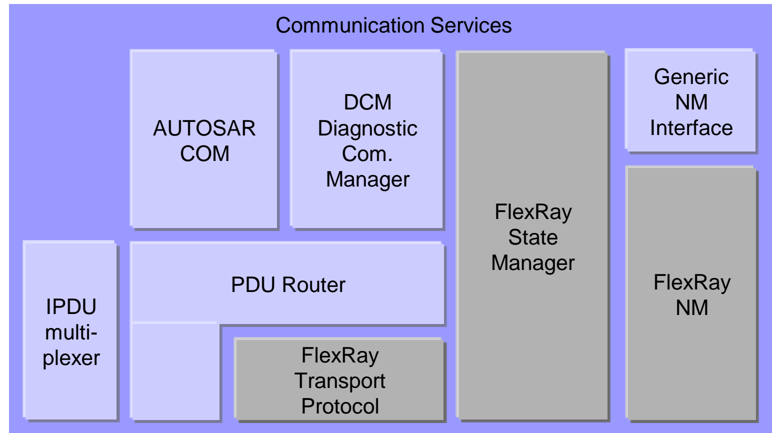
- AUTOSAR Confidential -

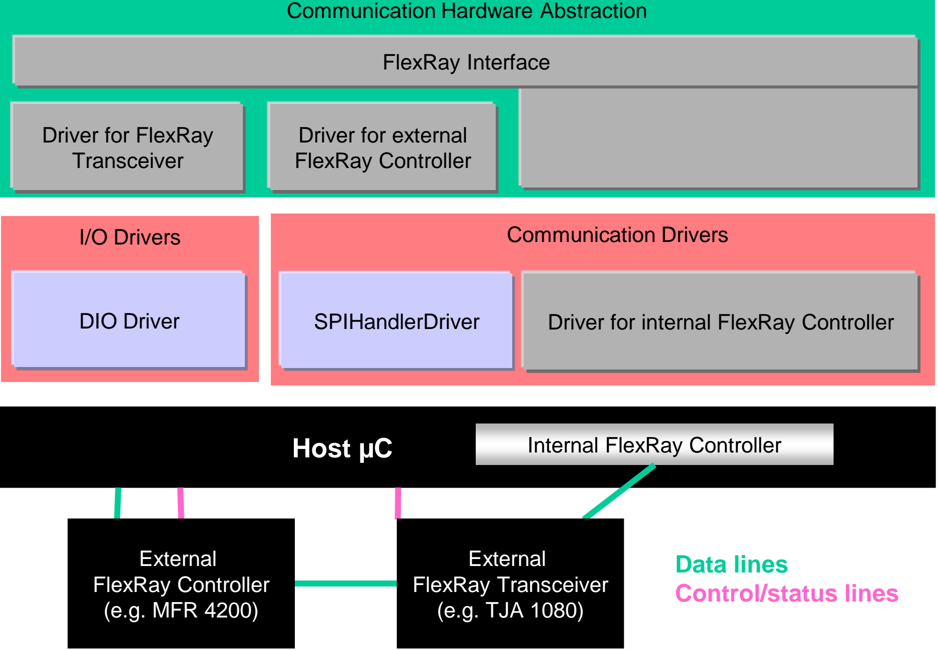
Part 3 – Contents of Software Layers
ID: 03-09 Scope: Communication Stack – FlexRay

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

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

Properties:
- Implementation: μC and ECU HW independent, partly dependent on FlexRay.
- AUTOSAR COM, 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 GW 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 will be 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.
- 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).





- AUTOSAR Confidential -

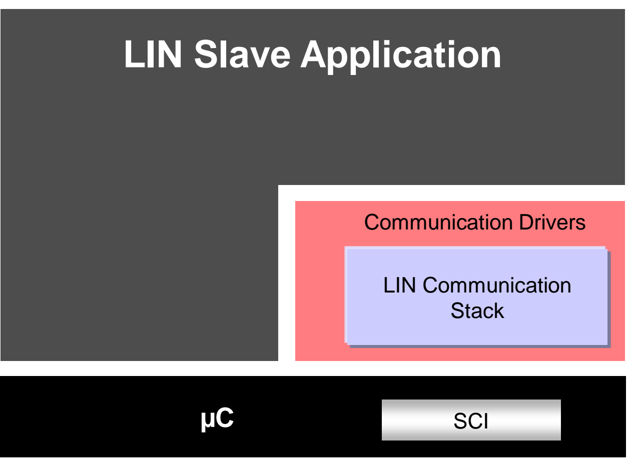
AUTOSAR

Part 3 – Contents of Software Layers
ID: 03-10 Scope: Communication Services – LIN Slave

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

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

That means: LIN Slaves can be connected as complete ECUs. But they are not forced to use the AUTOSAR SW Architecture.
Perhaps they can use some standard AUTOSAR modules (like EEPROM, DIO).
Reason: LIN slaves usually have very limited memory resources or are ASICs with „hard-coded“ logic.



Part 3 – Contents of Software Layers
ID: 03-12 Scope: Memory Services

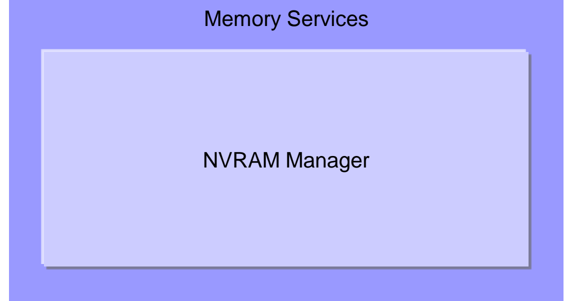
The **Memory Services** consist of one module, the NVRAM Manager. It is responsible for the management of non volatile data (read/write from different memory drivers). The application expects a RAM mirror as data interface for fast read access.

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

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

Memory Services
Communi-cation Services
I/O HW Abstraction
Complex Drivers
Onboard Dev. Abstraction
Memory HW Abstraction
COM HW Abstraction
Micro-controller Drivers
Memory Drivers
Communi-cation Drivers
I/O Drivers
Microcontroller (µC)

Example:

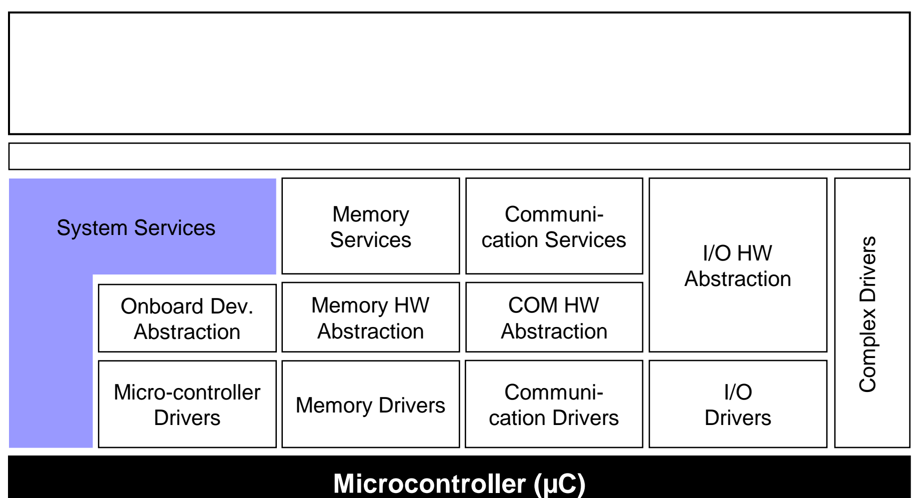


Part 3 – Contents of Software Layers
ID: 03-13 Scope: System Services

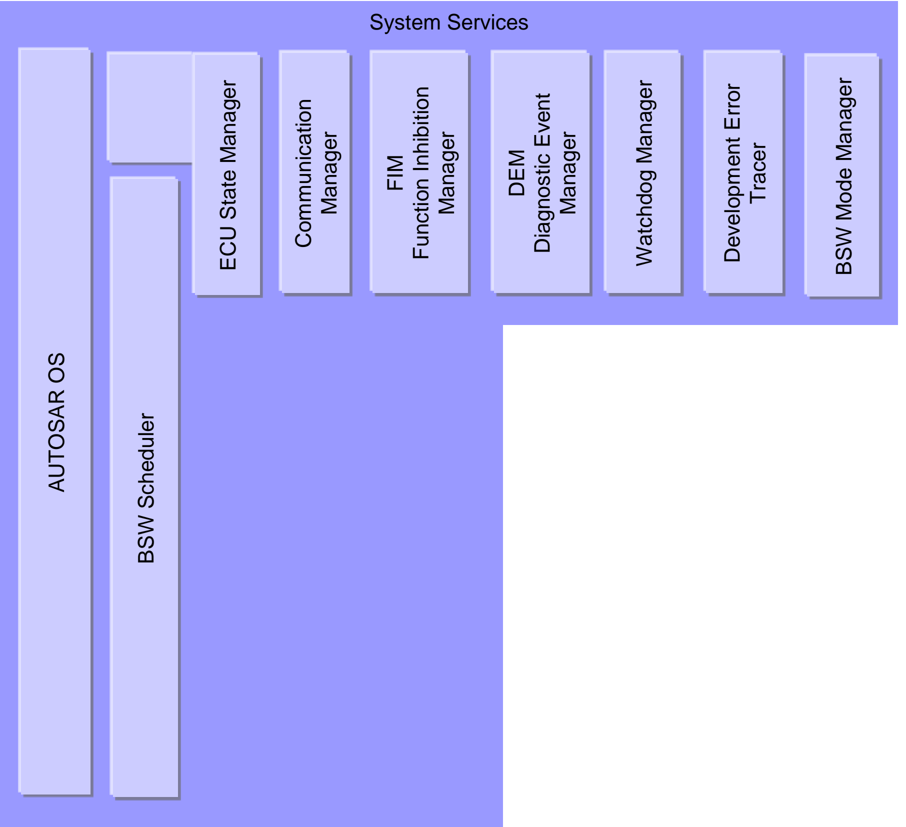
The **System Services** are a group of modules and functions which can be used by modules of all layers. Examples are Real Time Operating System (which includes timer services), Error Manager.
Some of these services are µC dependent (like OS), partly ECU hardware and application dependent (like ECU State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

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

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



Example:



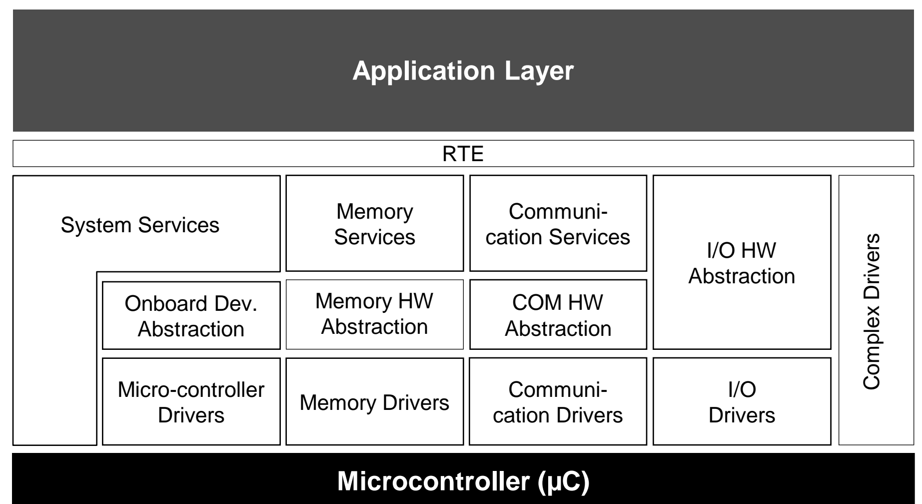
- AUTOSAR Confidential -

Part 3 – Contents of Software Layers
ID: 03-14 Scope: Sensor/Actuator AUTOSAR Software Components

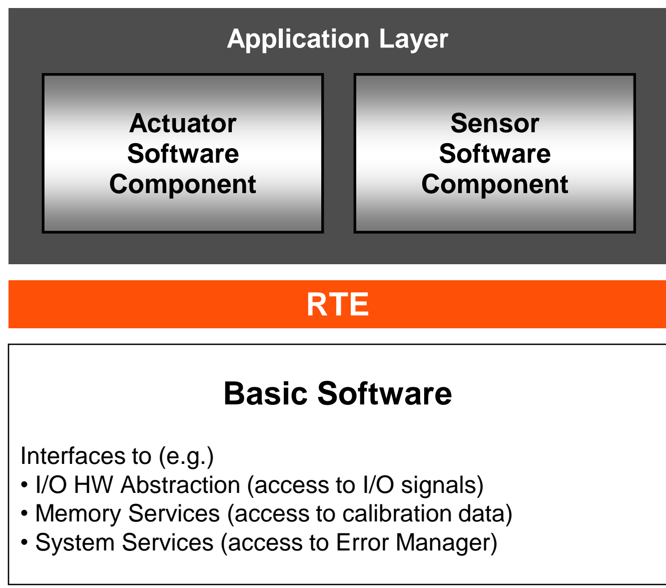
The **Sensor/Actuator AUTOSAR Software Component** is a specific type of AUTOSAR Software Component for sensor evaluation and actuator control. Though not belonging to the AUTOSAR Basic Software, it is described here due to its strong relationship to local signals. It has been decided to locate the Sensor/Actuator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

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

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



Example:



Part 4 – Interfaces
4.1 General Rules

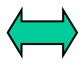
Part 4 – Interfaces
ID: 04-004 Type of Interfaces in AUTOSAR

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.

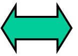
- AUTOSAR Confidential -

Part 4.1 – Interfaces: General Rules
ID: 04-002 General Interfacing Rules

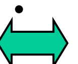
Horizontal Interfaces



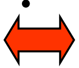
Services Layer: horizontal interfaces are allowed
Example: Error Manager saves fault data using the NVRAM manager



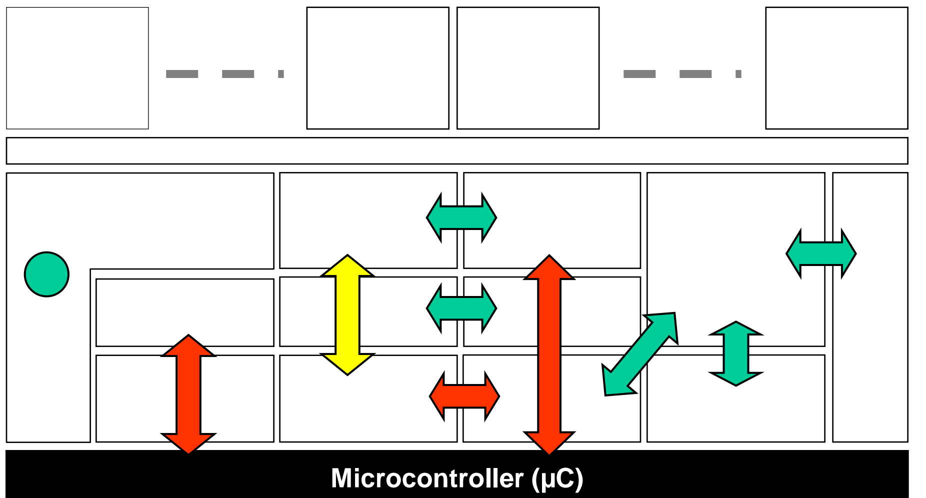
ECU Abstraction Layer: horizontal interfaces are allowed



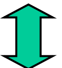
- A complex driver may use selected other BSW modules



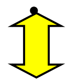
- µC Abstraction Layer: horizontal interfaces are not allowed. Exception: configurable notifications are allowed due to performance reasons.



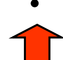
Vertical Interfaces



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.

- AUTOSAR Confidential -

Part 4.1 – Interfaces: General Rules
ID: 04-003 Layer Interaction Matrix

This matrix shows the possible interactions between AUTOSAR Basic Software layers

✓ "is allowed to use"
✗ "is not allowed to use"
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)

uses →

	System Services	Memory Services	Communication Services	Complex Drivers	I/O Hardware Abstraction	Onboard Device Abstraction	Memory Hardware Abstraction	Communication Hardware Abstraction	Microcontroller Drivers	Memory Drivers	Communication Drivers	I/O Drivers
AUTOSAR SW Components / RTE	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
System Services	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
Memory Services	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Communication Services	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
Complex Drivers	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I/O Hardware Abstraction	✓	✗	✗	✗	✓	✓	✗	✓	✓	✗	✓	✓
Onboard Device Abstraction	✓	✗	✗	✗	✗	✓	✗	✓	✓	✗	✓	✓
Memory Hardware Abstraction	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✗
Communication Hardware Abstraction	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗	✓	✓
Microcontroller Drivers	✓	✗	✗	✗	✓	✓	✗	✗	Δ	✗	✗	Δ
Memory Drivers	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Communication Drivers	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓
I/O Drivers	✓	✗	✗	✗	✓	✓	✗	✗	Δ	✗	✗	Δ

Part 4.1 – Interfaces: General Rules
ID: 04-005 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)

- AUTOSAR Confidential -

Part 4.1 – Interfaces: General Rules
ID: 04-006 Interfacing with Complex Drivers (2)

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

- The PDU Router as exclusive bus and protocol independent access point to the communication stack
- The CAN/FlexRay 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

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.

- AUTOSAR Confidential -

Part 4 – Interfaces
4.2 Interaction of Layers – Example “Memory”

Part 4.2 – Interfaces: Interaction of Layers – Example "Memory"
ID: 04-021 Introduction

The following pages explain using the example „memory":

- How do the software layers interact?
- How do the software interfaces look like?
- What is inside the ECU Abstraction Layer?
- How can abstraction layers be implemented efficiently?

Part 4.2 – Interfaces: Interaction of Layers – Example "Memory"
ID: 04-022 Example and First Look

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

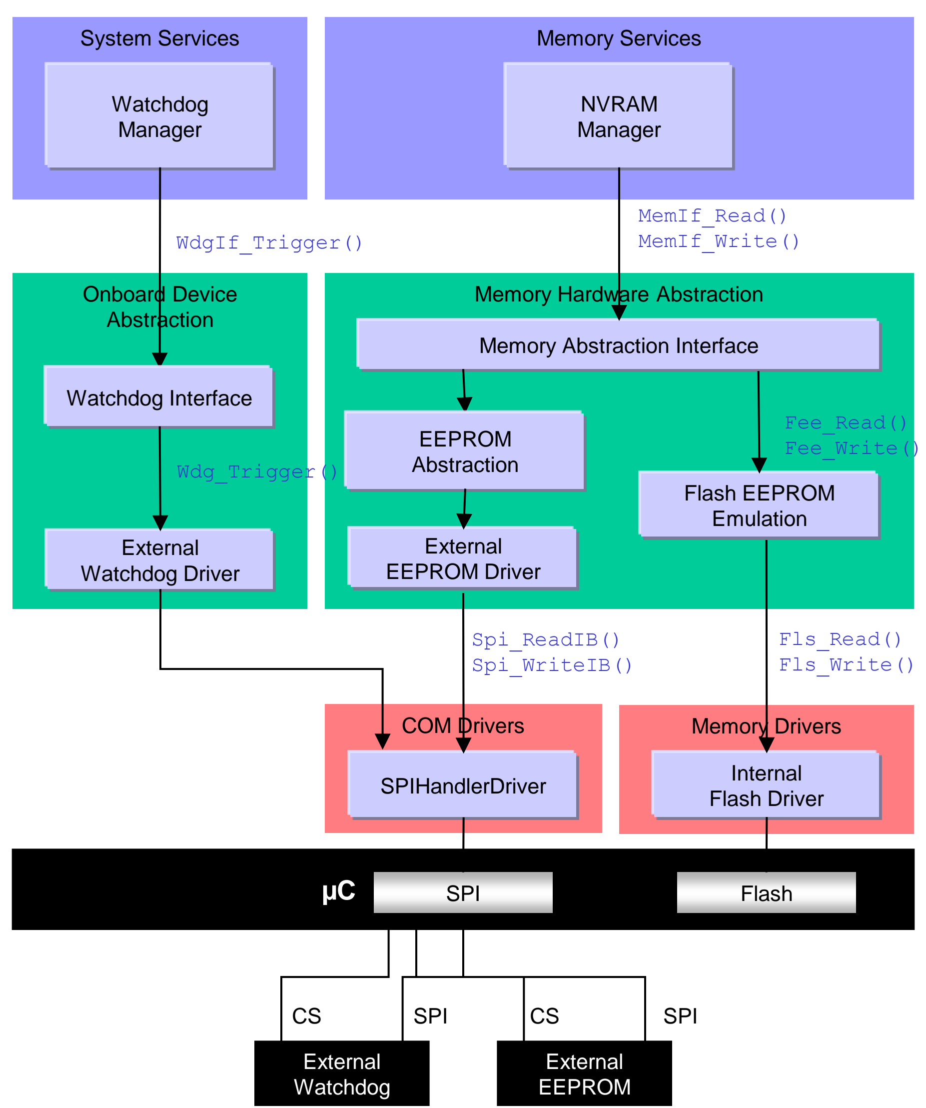
The ECU hardware includes an external EEPROM and an external watchdog connected to the microcontroller via the same SPI.

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“)



- AUTOSAR Confidential -
AUTOSAR

Part 4.2 – Interfaces: Interaction of Layers – Example "Memory"
ID: 04-023 Closer Look at Memory Hardware Abstraction

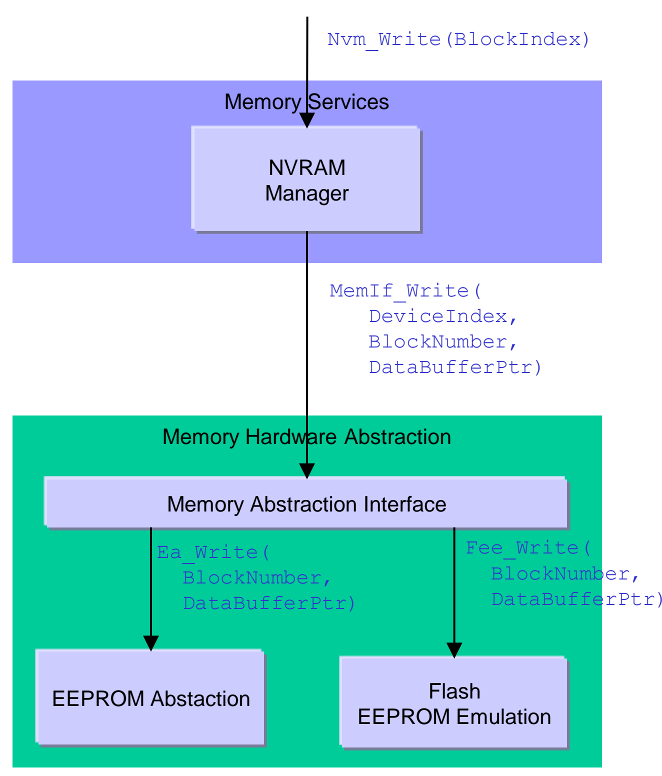
Architecture Description
The NVRAM Manager accesses drivers via the Memory Abstraction Interface. It addresses different memory devices using a device index.

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

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



- AUTOSAR Confidential -
AUTOSAR

Part 4.2 – Interfaces: Interaction of Layers – Example "Memory"
ID: 04-024 Implementation of Memory Abstraction Interface (1)

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

File MemIf.h:

```
#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.

- AUTOSAR Confidential -

Part 4.2 – Interfaces: Interaction of Layers – Example "Memory"
ID: 04-025 Implementation of Memory Abstraction Interface (2)

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.

- AUTOSAR Confidential -

Part 4.2 – Interfaces: Interaction of Layers – Example "Memory"
ID: 04-026 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
- The architectural targets and requirements are fulfilled

Part 4 – Interfaces
4.3 Interaction of Layers – Example “Communication”

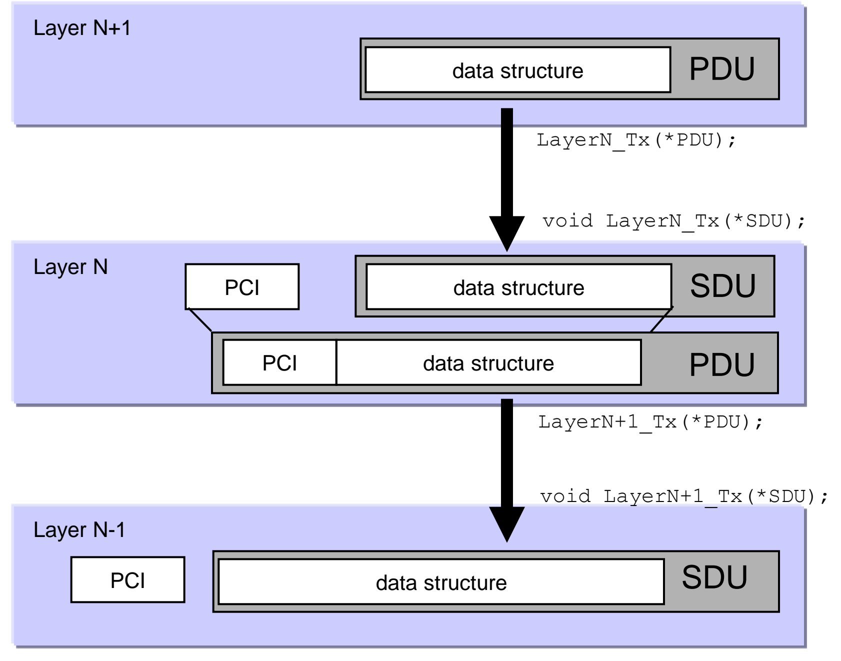
Part 4.3 – Interfaces: Interaction of Layers – Example "Communication"
ID: 04-051 PDU Flow through the Layered Architecture

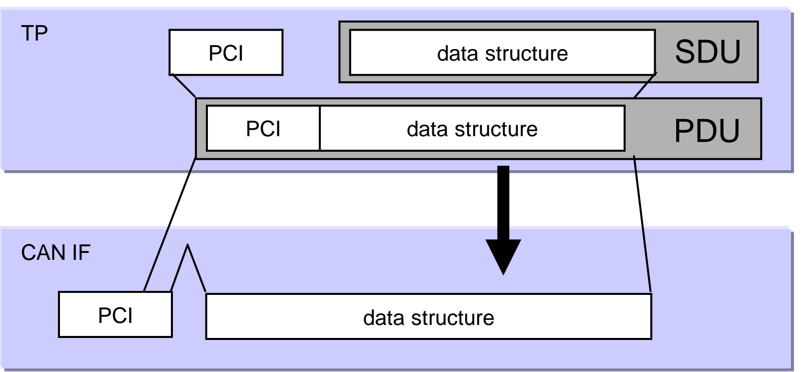
Explanation of terms:

- **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.





- AUTOSAR Confidential -

Part 4.3 – Interfaces: Interaction of Layers – Example “Communication”
ID: 04-052 SDU and PDU Naming Conventions

Naming of PDUs and SDUs respects the following rules:

For PDU:

<***bus prefix***> <***layer prefix***> - PDU

For SDU

<***bus prefix***> <***layer prefix***> - SDU

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

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

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

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

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

Part 4.3 – Interfaces: Interaction of Layers – Example “Communication”
ID: 04-050 Generic Gateway and COM Layer Structure

Routing 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 Gateway
  - Synchronization of Network States of different communication channels connected to an ECU via the network managements handled by the NM Gateway
- Communication State Managers
  - Start and Shutdown the hardware units of the communication systems via the interfaces.
  - Control PDU groups

RTE
Signals
Communication Manager
Generic NM interface
NM Coordinator
IPDU multi-plexer
AUTOSAR COM
DCM Diagnostic Communication Manager
I-PDU
I-PDU
I-PDU
FlexRay State Manager
CAN State Manager
LIN State Manager
NM Module
PDU Router
I-PDU[1]
Í-PDU
I-PDU[1]
I-PDU
FlexRay TP
CAN TP
N-PDU
N-PDU
Communication HW Abstraction
FlexRay Interface
CAN Interface
LIN Interface (incl. LIN TP)
L-PDU
L-PDU
L-PDU
Communication Drivers
FlexRay Driver
CAN Driver
LIN Low Level Driver

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

ID: 04-070 – Layered Software Architecture

Part 4 – Interfaces
4.4 Interaction of Layers –
Example "ECU State Manager"

- AUTOSAR Confidential -

Part 4.4 – Interfaces: Interaction of Layers – Example “ECU State Manager”
ID: 04-071 Interaction with ECU State Manager

RTE

Request/release
RUN

Service Layer (except communication stack)

Communication Services

Post build selectable Configuration tables

1
2
3

DEM
ComM
WdgM
…

DEM
ComM
WdgM
…

DEM
ComM
WdgM
…

ECU State Manager

Report Error

initializes

DEM

DET

Watchdog Manager

Communication Manager

NVRAM Manager

Job End Notification

Report Error

Store Errors

CAN State Manager

FlexRay State Manager

LIN State Manager

BSW Scheduler

Call main functions
and access
restricted areas

initializes

initializes

initializes

initializes

ECU Abstraction Layer

Microcontroller Abstraction Layer

This figure does not show all interactions between all modules. It is a discussion base only.

Part 5 – Configuration

Part 5 – Configuration
ID: 05-000 Overview

The AUTOSAR Basic Software supports the following configuration classes:

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

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

3. **Post build time**
   - Loadable constant data outside the module. Very similar to [2], but the data is located in a specific memory segment that allows reloading (e.g. reflashing in ECU production line)
   - Single or multiple configuration sets can be provided. In case that multiple configuration sets are provided, the actually used configuration set is to be specified at runtime.

In many cases, the configuration parameters of one module will be of different configuration classes.
Example: a module providing post build time configuration parameters will still have some parameters that are pre compile time configurable.

Part 5 – Configuration
ID: 05-001 Pre Compile Time

Use cases

Pre compile time configuration would be chosen for

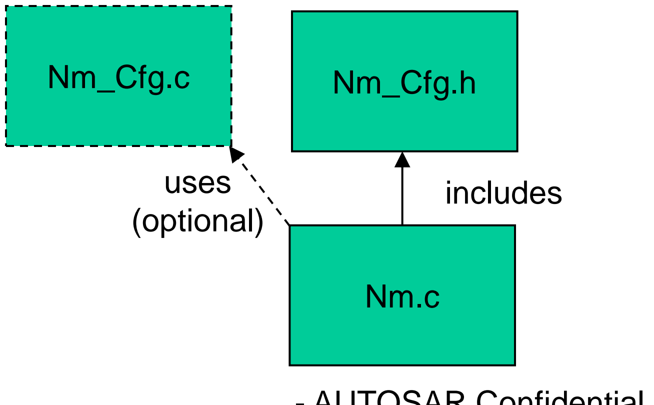
- Enabling/disabling optional functionality
  This allows to exclude parts of the source code that are not needed
- Optimization of performance and code size
  Using `#defines` results in most cases in more efficient code than access to constants or even access to constants via pointers.
  Generated code avoids code and runtime overhead.

Restrictions

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

Required implementation

Pre compile time configuration shall be done via the module‘s two configuration files (*_Cfg.h, *_Cfg.c) and/or by code generation:
• *_Cfg.h stores e.g. macros and/or `#defines`
• *_Cfg.c stores e.g. constants



- AUTOSAR Confidential -

Part 5 – Configuration
ID: 05-004 Pre Compile Time

Example 1: Enabling/disabling functionality

File Spi_Cfg.h:

```
#define SPI_DEV_ERROR_DETECT    ON
```

File Spi_Cfg.c:

```
const uint8 myconstant = 1;
```

File Spi.c (available as source code):

```
#include "Spi_Cfg.h"     /* for importing the configuration parameters */

external const uint8 myconstant;

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

Part 5 – Configuration
ID: 05-002 Pre Compile Time

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 Dem_FLS_E_ERASE_FAILED           1U
#define Dem_FLS_E_WRITE_FAILED           2U
#define Dem_FLS_E_READ_FAILED            3U
#define Dem_FLS_E_UNEXPECTED_FLASH_ID    4U
#define Dem_NVM_E_REQ_FAILED             5U
#define Dem_CANSM_E_BUSOFF_NETWORK_5     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_ReportErrorStatus(Dem_NVM_E_REQ_FAILED, DEM_EVENT_STATUS_PASSED);
```

Part 5 – Configuration
ID: 05-003 Link Time

Use cases

Link time configuration would be chosen for

- Configuration of modules that are only available as object code
  (e.g. IP protection or warranty reasons)
- Selection of configuration set after compilation but before linking.

Required implementation

1. One configuration set, no runtime selection
   Configuration data shall be captured in external constants. These external constants are located in a separate file. The module has direct access to these external constants.

Part 5 – Configuration
ID: 05-005 Link Time

Example 1: Event IDs reported to the DEM by a module (CAN Interface) that is available as object code only
XML configuration file of the CAN Interface:
Specifies that it needs the event symbol `CANIF_E_INVALID_DLC` 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 Dem_FLS_E_UNEXPECTED_FLASH_ID    1U
#define Dem_NVM_E_REQ_FAILED             2U
#define Dem_CAN_E_TIMEOUT                3U
#define Dem_CANIF_E_INVALID_DLC          4U
...
```

File CanIf_Lcfg.c:

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

const Dem_EventIdType CanIf_InvalidDlc = Dem_CANIF_E_INVALID_DLC;
```

File CanIf.c (available as object code):

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

Dem_ReportErrorStatus(CanIf_InvalidDlc, DEM_EVENT_STATUS_FAILED);
```

Note: the complete include file structure with all forward declarations is not shown here to keep the example simple.

Part 5 – Configuration
ID: 05-006 Link Time

Example 1: Event IDs reported to the DEM by a module (CAN Interface) that is available as object code only

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

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

Part 5 – Configuration
ID: 05-007 Post Build Time

Use cases

Post build time configuration would be chosen for

- Configuration of data where only the structure is defined but the contents not known during ECU build time
- Configuration of data that is likely to change or has to be adapted after ECU build time
  (e.g. end of line, during test & calibration)
- Reusability of ECUs across different product lines (same code, different configuration data)

Restrictions

- Implementation requires dereferencing which has impact on performance, code and data size

Required implementation

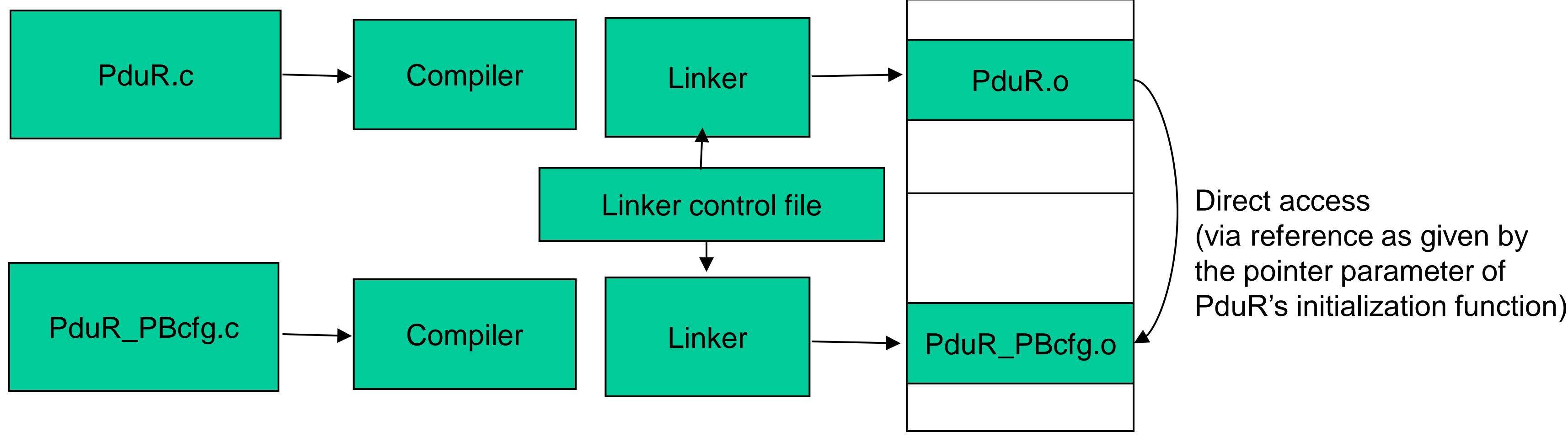
1. One configuration set, no runtime selection (**loadable**)
   Configuration data shall be captured in external constant structs. These external structs are located in a separate memory segment that can be individually reloaded.

2. 1..n configuration sets, runtime selection possible (**selectable**)
   Configuration data shall be captured within external constant structs. These configuration structures are located in one separate file. The module gets a pointer to one of those structs at initialization time. The struct can be selected at each initialization.

- AUTOSAR Confidential -

Part 5 – Configuration
ID: 05-010 Post Build Time

Example 1 (Post Build Time loadable)
If the configuration data is fix in memory size and position, the module has direct access to these external structs.



Part 5 – Configuration
ID: 05-008 Post Build Time

Required implementation 2: Configuration of CAN Driver that is available as object code only; multiple configuration sets can be selected during initialization time.

File Can_PBcfg.c:

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

File EcuM.c:

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

File Can.c (available as object code):

```
#include "Can.h"     /* for getting Can_ConfigType      */

void Can_Init(Can_ConfigType* Config)
{
  /* write the init data to the CAN HW */
};
```

Compiler

Linker

Binary file

- AUTOSAR Confidential -

AUTOSAR

Part 5 – Configuration
ID: 05-009 Variants

Different use cases require different kinds of configurability:

Example use cases:

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

To allow the implementation of such different use cases in each BSW module for each module, up to 3 variants can be specified:

- A variant is a dedicated assignment of the configuration parameters of a module to configuration classes
- Within a variant a configuration parameter can be assigned to only ONE configuration class
- Within a variant a configuration class for different configuration parameters can be different (e.g. Pre-Compile for development error detection and post-build for reprogrammable PDU routing tables
- It is possible and intended that specific configuration parameters are assigned to the same configuration class for all variants (e.g. development error detection is in general pre-compile time configurable).

- AUTOSAR Confidential -

Part 5 – Configuration
ID: 05-011 Memory Layout Example: Postbuild Loadable (PBL)

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 modules data is agreed locally (in the module) only

1. The module (‘xx’, ‘yy’) knows its own start address (to enable the implementer to allocate data section)
2. Only the module (‘xx’, ‘yy’) knows the internals of its own configuration

For details, see Chapter “Post build implementation” in “AUTOSAR_SWS_C_ImplementationRules.pdf”

Part 5 – Configuration
ID: 05-012 Memory Layout Example: Postbuild Multiple Selectable (PBM)

	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
	…	
RL	0x8010	&xx_configuration = …
	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 PBL

For details, see Chapter “Post build implementation” in “AUTOSAR_SWS_C_ImplementationRules.pdf”

Part 6 – Scheduling

Part 6 – Scheduling
ID: 06-001 Basic Scheduling Concepts of the BSW

- **BSW Scheduling shall**
  - Assure correct timing behavior of the BSW, i.e., correct interaction of **all** BSW modules with respect to time
  - Be used to apply data consistency mechanisms

- **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 implemented by the ECU/BSW integrator**
  - 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)

- **Restrict the usage of OS functionality**
  - Only the Schedule Module shall use OS objects or OS services
    (exceptions: EcuM and services: `GetCounterValue` and `GetElapsedCounterValue` of OS)
  - 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

- AUTOSAR Confidential -

Part 6 – Scheduling
ID: 06-003 Scheduling Objects and Triggers

BSW Scheduling objects

- Main functions
  - n per module
  - located in all layers

BSW Events

- RecurringEvent
- SporadicEvent

Triggers

- Main functions
  - Can be triggered in all layers by
    - RecurringEvents
    - SporadicEvents

RTE

Zzz_MainFunction_Aaa

Yyy_MainFunction_Aaa

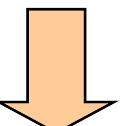
Xxx_Isr_Yyy

Microcontroller

- AUTOSAR Confidential -

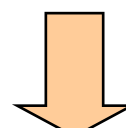
Part 6 – Scheduling
ID: 06-004 Transformation Process (1)

Logical Architecture (Model)
- Ideal concurrency
- Unrestricted resources
- Only real data dependencies



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

Technical Architecture (Implementation)
- Restricted concurrency
- Restricted resources
- Real data dependencies
- Dependencies given by restrictions



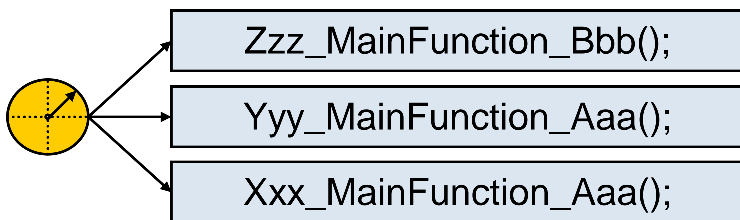
- OS objects
  - Tasks
  - ISRs
  - Alarms
  - Resources
  - OS services
- Sequences of scheduling objects within tasks
- Sequences of tasks
- ...

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

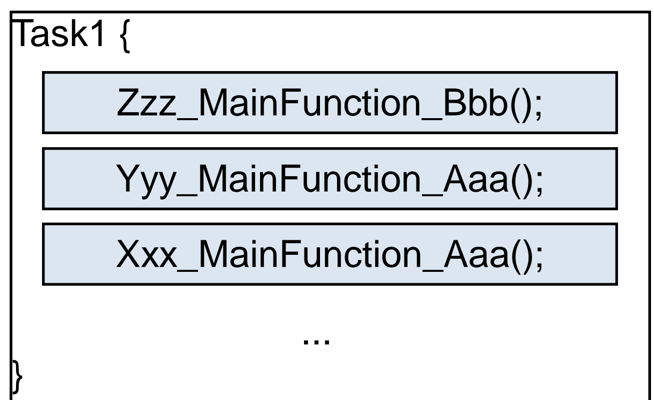
- AUTOSAR Confidential -

Part 6 – Scheduling
ID: 06-006 Transformation Process – Example 1

Logical Architecture (Model)

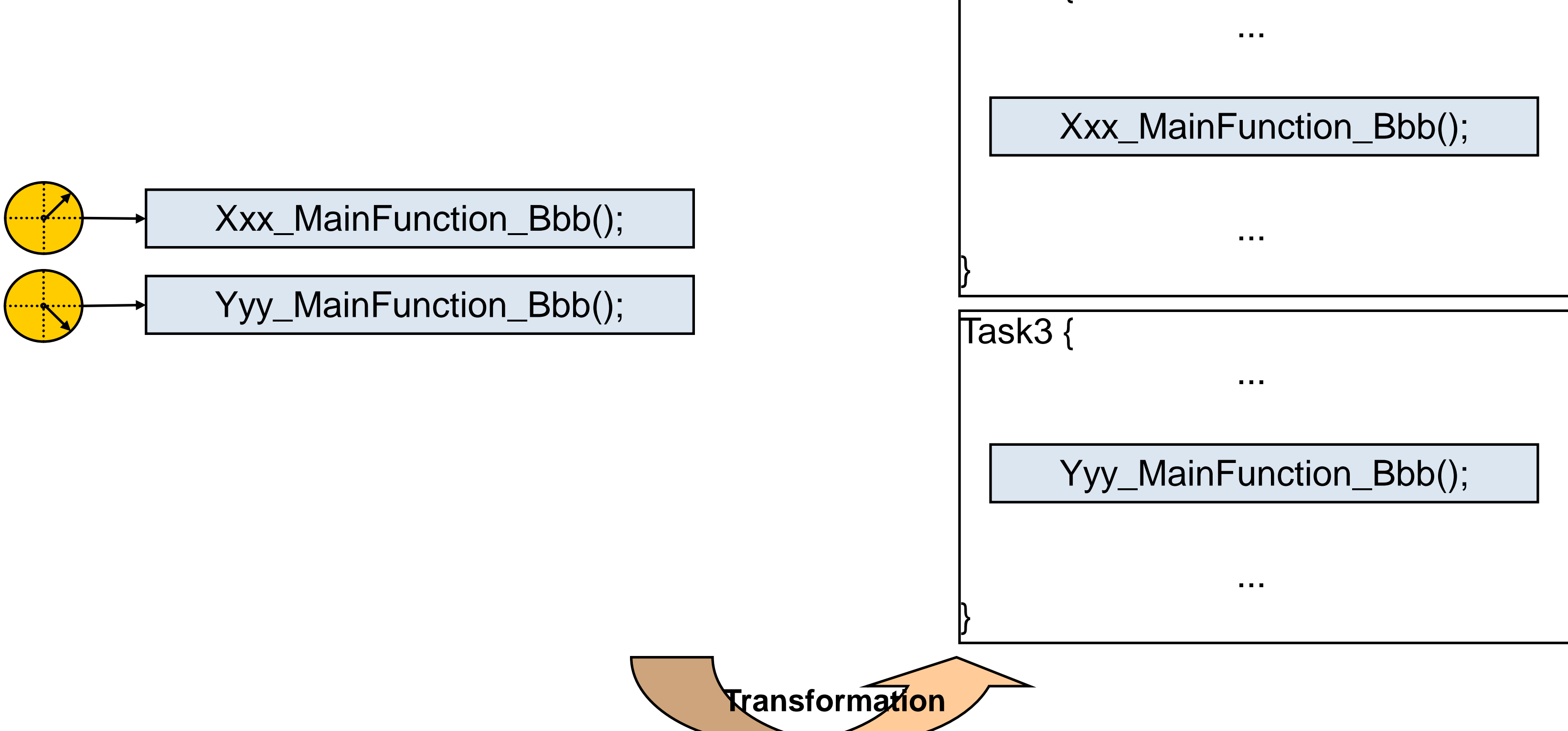


Technical Architecture (Schedule Module SchM)



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

Part 6 – Scheduling
ID: 06-007 Transformation Process – Example 2



- Mapping of scheduling objects to OS Tasks

- AUTOSAR Confidential -

Part 6 – Scheduling
ID: 06-008 Data Consistency – Motivation

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

Logical Architecture (Model)

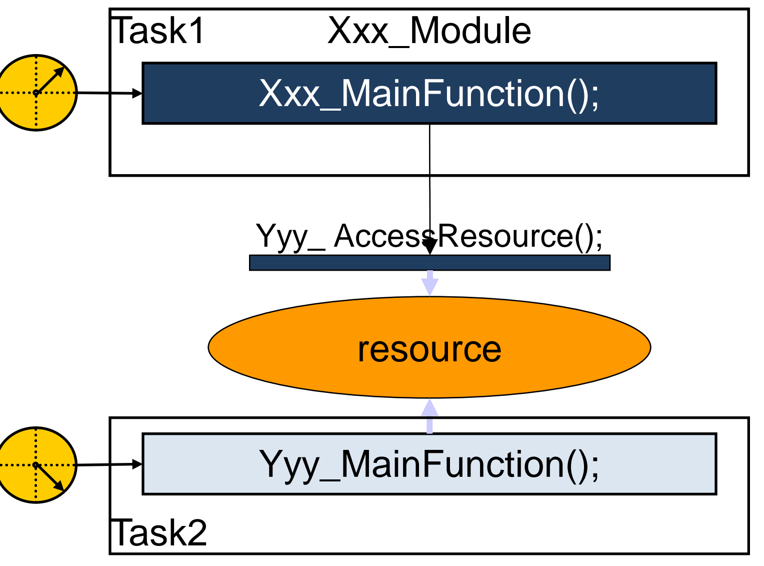
Technical Architecture (Schedule Module SchM)

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

- AUTOSAR Confidential -

Part 6 – Scheduling
ID: 06-009 Data Consistency – Example 1 – "Critical Sections" Approach

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



Implementation of Schedule Module SchM

```
#define SchM_Enter_<mod>(XYZ)   DisableAllInterrupts
#define SchM_Exit_<mod>(XYZ)    EnableAllInterrupts
```

```
Yyy_AccessResource() {
    ...
    SchM_Enter_xxx_(XYZ)
    <access_to_shared_resource>
    SchM_Exit_xxx(XYZ)
    ...
}
```

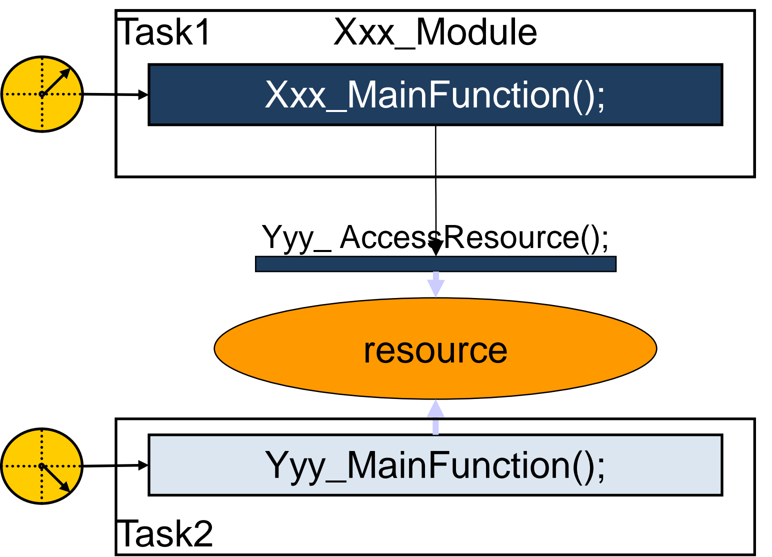
```
Yyy_MainFunction() {
    ...
    SchM_Enter_yyy_(XYZ)
    <access_to_shared_resource>
    SchM_Exit_yyy(XYZ)
    ...
}
```

- Data consistency is ensured by
  - Interrupt blocking

- AUTOSAR Confidential -

Part 6 – Scheduling
ID: 06-009 Data Consistency – Example 2 – “Critical Sections” Approach

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



Implementation of Schedule Module SchM

```
#define SchM_Enter_<mod>(XYZ) /* nothing required */
#define SchM_Exit_<mod>(XYZ)  /* nothing required */
```

```
Yyy_AccessResource() {
    ...
    SchM_Enter_xxx_(XYZ)
    <access_to_shared_resource>
    SchM_Exit_xxx(XYZ)
    ...
}
```

```
Yyy_MainFunction() {
    ...
    SchM_Enter_yyy_(XYZ)
    <access_to_shared_resource>
    SchM_Exit_yyy(XYZ)
    ...
}
```

- Data consistency is ensured by
  - Sequence

Part 7 – Implementation Conformance Classes

Part 7 – Implementation Conformance Classes
ID: 07-001 ICC3

Application Layer

AUTOSAR Runtime Environment (RTE)

System Services
AUTOSAR OS
BSW Scheduler
ECU State Manager
Communication Manager
Function Inhibition Manager
Diagnostic Event Manager
Watchdog Manager
Development Error Tracer
BSW Mode Manager

Memory Services
NVRAM Manager

Communication Services
COM
DCM
SM
NM If
IPDU
PduR
TP
NM

I/O Hardware Abstraction
I/O Signal Interface
Driver for ext. ADC ASIC
Driver for ext. I/O ASIC

Complex Drivers

Onboard Device Abstraction
WdgIf

Memory Hardware Abstraction
MemIf
EA
Fee

Communication Hardware Abstraction
xxx Interface
Trcv.
ext. Drv

Microcontroller Drivers
GPT Driver
Watchdog Driver
MCU Driver
Core Test

Memory Drivers
RAM Test
internal Flash Driver
internal EEPROM Driver

Communication Drivers
SPIHandler Driver
LIN Driver
CAN Driver
FlexRay Driver

I/O Drivers
ICU Driver
PWM Driver
ADC Driver
DIO Driver
PORT Driver

Microcontroller

Not all ICC3 modules shown

ICC3 module

functional groups

- AUTOSAR Confidential -

Part 7 – Implementation Conformance Classes
ID: 07-002 ICC2

Application Layer

AUTOSAR Runtime Environment (RTE)*

OS

SchM: SchM

Com Services: COM, PDU Router

Mode: EcuM, NM If, ComM

IPDUM*: IPDUM

Watch-dog: WdgM

Memories: NvM

Debug: DET

Diagnostics: FIM, DEM, DCM

I/O Hardware Abstraction

Complex Drivers

CAN: CAN St Mgr, CAN TP, CAN NM, CAN Interface, CAN Driver

FlexRay: FR St Mgr, FR TP, FR NM, FR Interface, FlexRay Driver

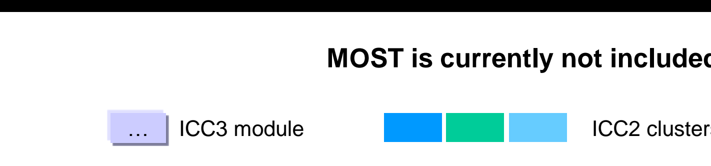
LIN: LIN St Mgr, LIN NM, LIN Interface, LIN Driver

FlexRay Transceiver, CAN Transceiver, WdgIf, MemIf, EA, Fee

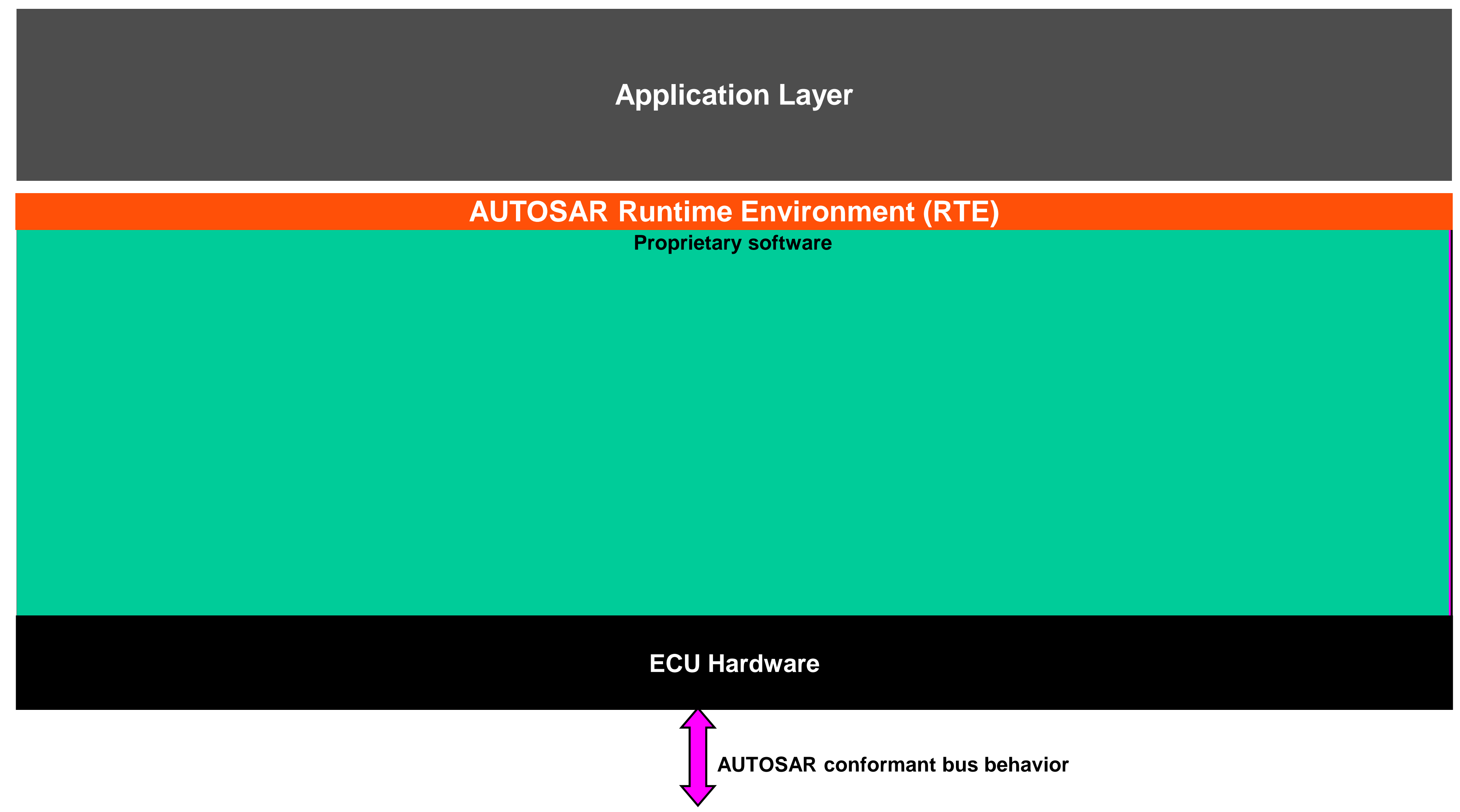
MCU Driver, Core Test, RAM Test, Watchdog Driver, EEPROM Driver, Flash Driver, SPIHandlerDriver, GPT Driver, PORT Driver, ICU Driver, PWM Driver, ADC Driver, DIO Driver

ECU Hardware

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 conformance.



Part 7 – Implementation Conformance Classes
ID: 07-003 ICC1



Part 8 – Integration and Runtime aspects

- AUTOSAR Confidential -

ID: 08-007 – Layered Software Architecture

Part 8 – Integration and Runtime aspects
8.1 Error Handling and Reporting Concept

Part 8.1 – Error Handling and Reporting Concept
ID: 08-009 Error Classification (1)

Types of errors

Hardware errors / failures

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

Software errors

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

System errors

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

- AUTOSAR Confidential -

Part 8.1 – Error Handling and Reporting Concept
ID: 08-010 Error Classification (2)

Time of error occurrence according to product life cycle

Development

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

Production / series

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

Influence of error on system

Severity of error (impact on the system)
- No influence
- Functional degradation
- Loss of functionality

Failure mode in terms of time
- Permanent errors
- Transient / sporadic errors

- AUTOSAR Confidential -

Part 8.1 – Error Handling and Reporting Concept
ID: 08-011 Error Reporting – Alternatives

Each basic software module distinguishes between two types of errors:

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

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

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

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

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

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

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

- AUTOSAR Confidential -

Part 8.1 – Error Handling and Reporting Concept
ID: 08-013 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 Development Error Tracer (see 08-014).

Example: services of EEPROM driver
Success: EEP_E_OK
General failure (service not accepted): EEP_E_NOT_OK
Write Operation to EEPROM was not successful: EEP_E_WRITE_FAILED

Part 8.1 – Error Handling and Reporting Concept
ID: 08-014 Error Reporting – Introduction

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

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

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

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

The detection and reporting of development errors to the Development Error Tracer can be statically switched on/off per module (preprocessor switch or two different object code builds of the module).

Part 8.1 – Error Handling and Reporting Concept
ID: 08-017 Error Reporting – Diagnostic Event Manager

API
The **Diagnostic Event Manager** has semantically the following API:
`Dem_ReportErrorStatus(EventId, EventStatus)`

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

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

Process:
1. Each BSW Module declares all production code relevant error variables it needs as “extern”
2. Each BSW Module stores all error variables that it needs in the ECU configuration description (e.g. `NVM_E_REQ_FAILED`)
3. The configuration tool of the Diagnostic Event Manager parses the ECU configuration description and generates a single file with global constant variables that are expected by the SW modules (e.g. `const Dem_EventIdType Dem_NVM_E_REQ_FAILED=7;` or `#define Dem_NVM_E_REQ_FAILED ((Dem_EventIdType)7)`
4. The reaction to the errors is also defined in the Error Manager configuration tool. This configuration is project specific.

Part 8.1 – Error Handling and Reporting Concept
ID: 08-018 Error Reporting – Development Error Tracer (1)

API

The **Development Error Tracer** has syntactically the following API:

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

Error numbering concept

`ModuleId (uint16)`

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

`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.

- AUTOSAR Confidential -

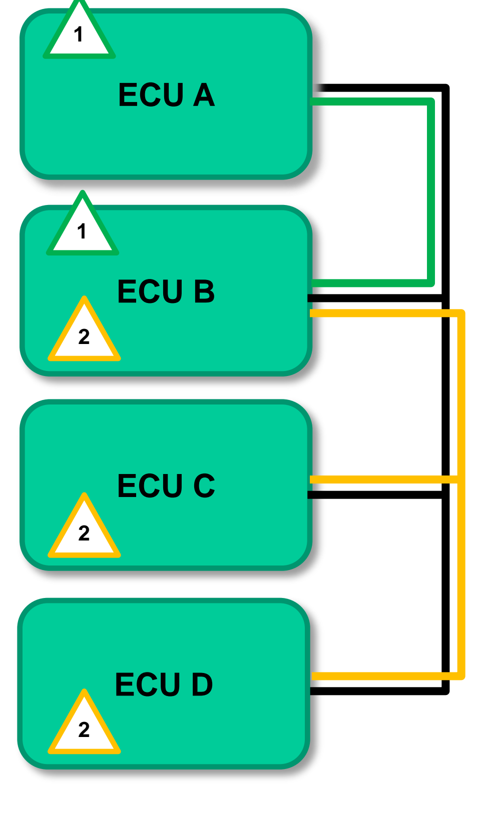
Part 8 – Integration and Runtime aspects
8.2 Partial Networking Concept

Part 8.2 – Partial Networking Concept Introduction

- The goal of the Partial Networking Concept is to provide mechanisms for power saving, especially while bus communication is active (e.g. charging or clamp 15 active).
- 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.
- Partial Networking uses NM messages to communicate the request/release information of a partial network cluster between the participating ECUs.

- AUTOSAR Confidential -

Part 8.2 – Partial Networking Concept
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.

- AUTOSAR Confidential -

Part 8.2 – Partial Networking Concept
Conceptual terms

- **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.

- AUTOSAR Confidential -

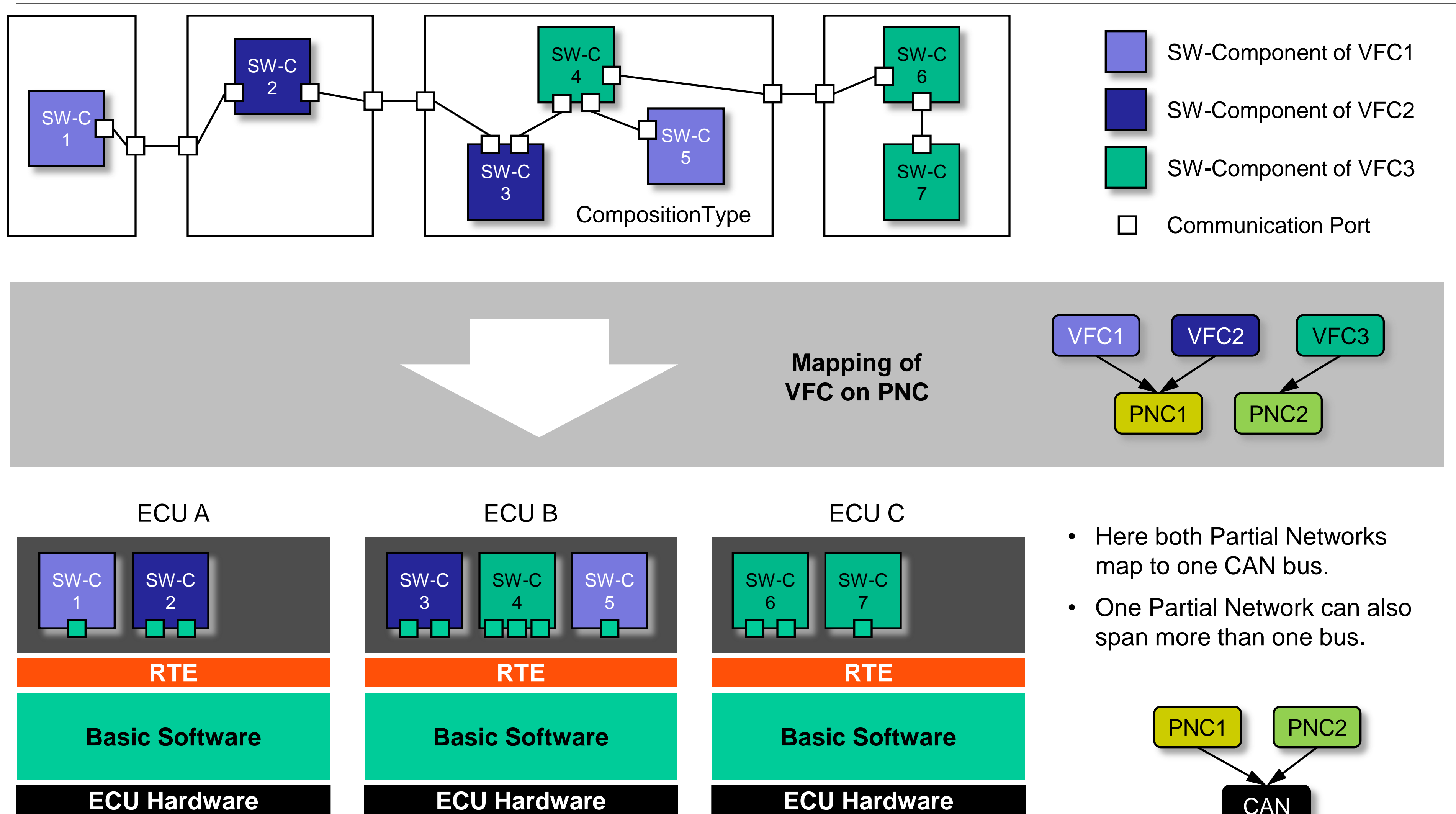
Part 8.2 – Partial Networking Concept Restrictions

- 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 PN information 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 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.

- AUTOSAR Confidential -

Part 8.2 – Partial Networking Concept
Mapping of Virtual Function Cluster to Partial Network Cluster



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

Part 8.2 – Partial Networking Concept Involved modules – Solution for CAN

Application Layer

SW-C

SW-C

RTE

- Coordination of I-PDU group switching
- Start / stop I-PDU-groups

Mode request

or

ComM_User Request

- VFC to PNC to channel translation
- PNC management (request / release of PNCs)
- Indication of PN states

System Services
Communication Services

BswM

ComM_UserRequest

ComM

PNC states

I-PDU GroupSwitch

Network Request

Request ComMode

PNC request/release information

- Exchange PNC request / release information between NM and ComM via NM user data
- Enable / disable I-PDU-groups

COM

NmIf

PduR

Trigger Transmit

CanNm

CanSM

Communication Hardware Abstraction

CanIf

- Filter incoming NM messages
- Collect internal and external PNC requests
- Send out PNC request infocmation in NM user data
- Spontaneous sending of NM messages on PNC startup

CanTrcv

- AUTOSAR Confidential -