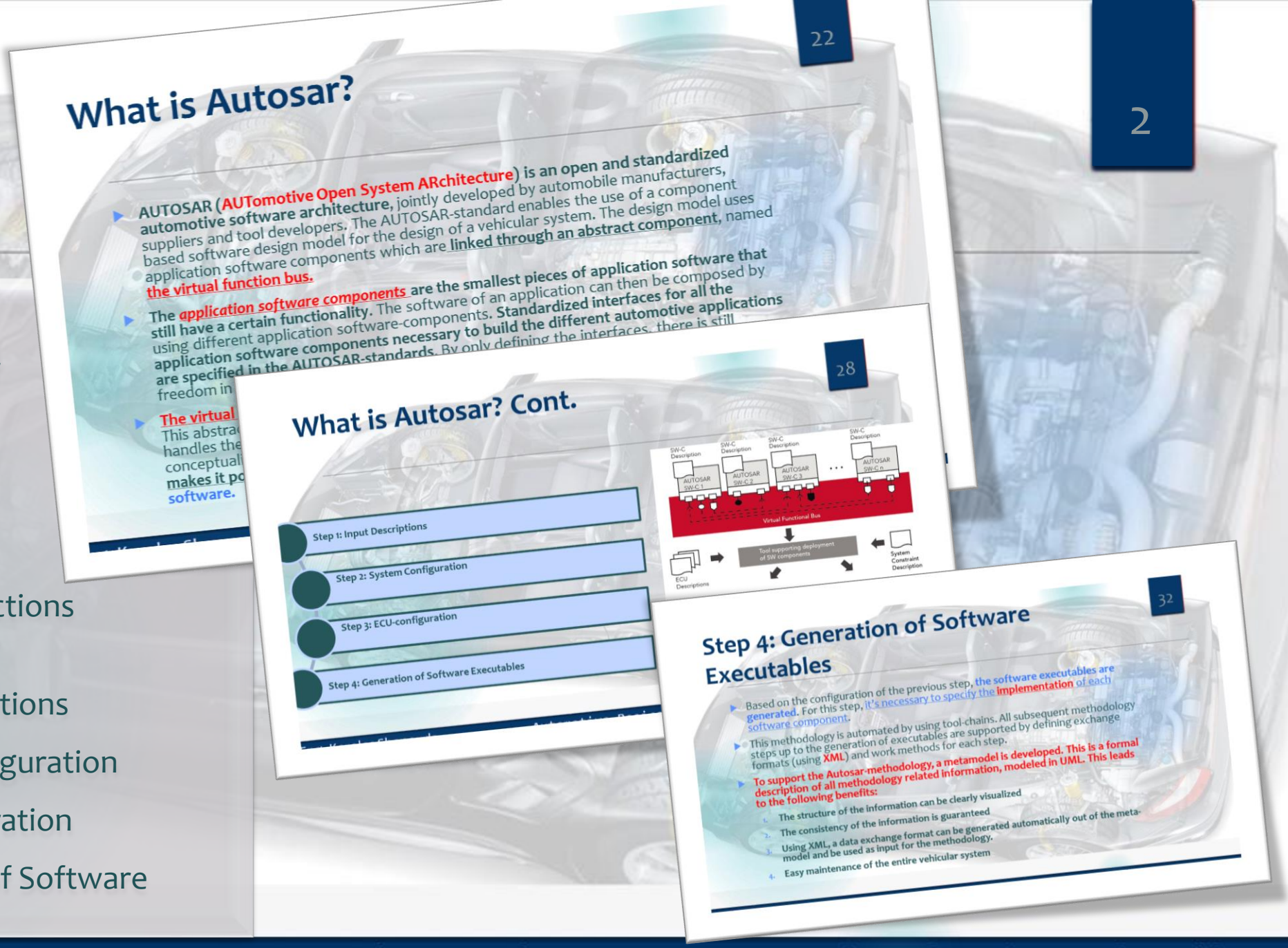
Embedded Automotive Basics

ENG. KEROLES SHENOUDA

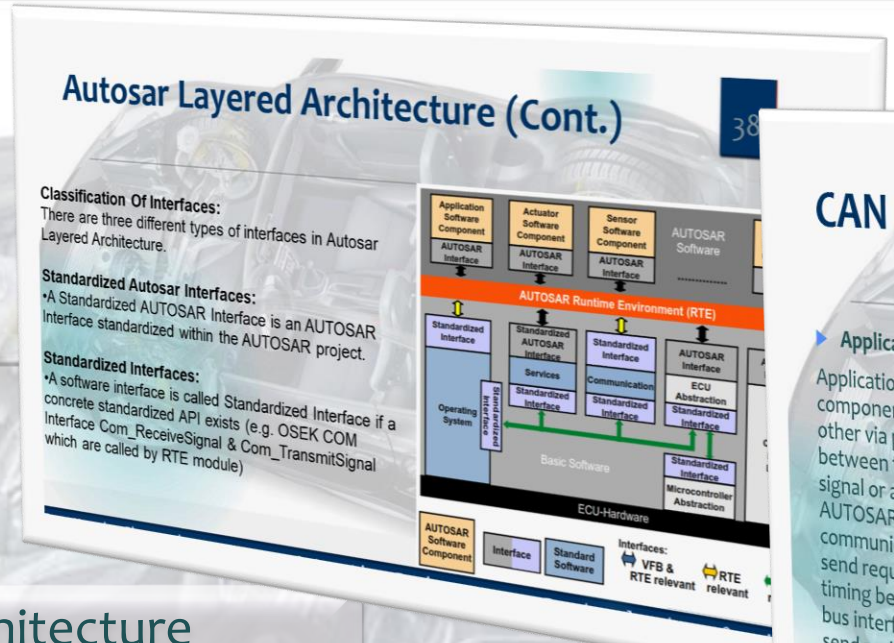
Index 1

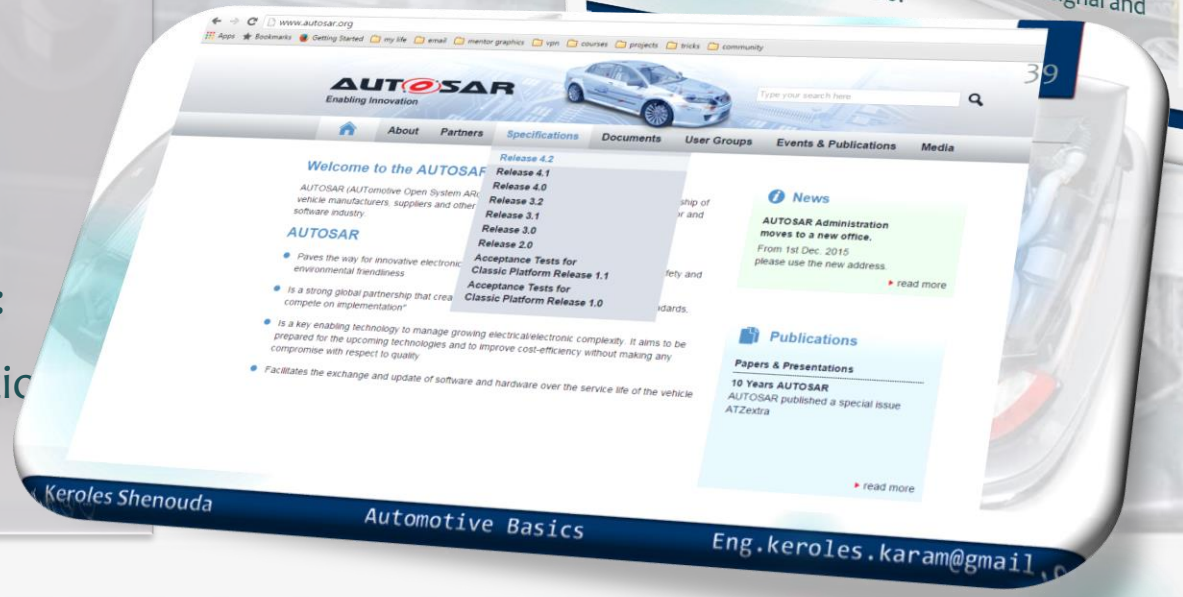
- Introduction
  - Automotive industry
  - Automotive Cycle
- What is AUTOSAR
  - Benefits of Autosar:
  - How are vehicle functions implemented today?
  - Step 1: Input Descriptions
  - Step 2: System Configuration
  - Step 3: ECU-configuration
  - Step 4: Generation of Software Executables



Index 2

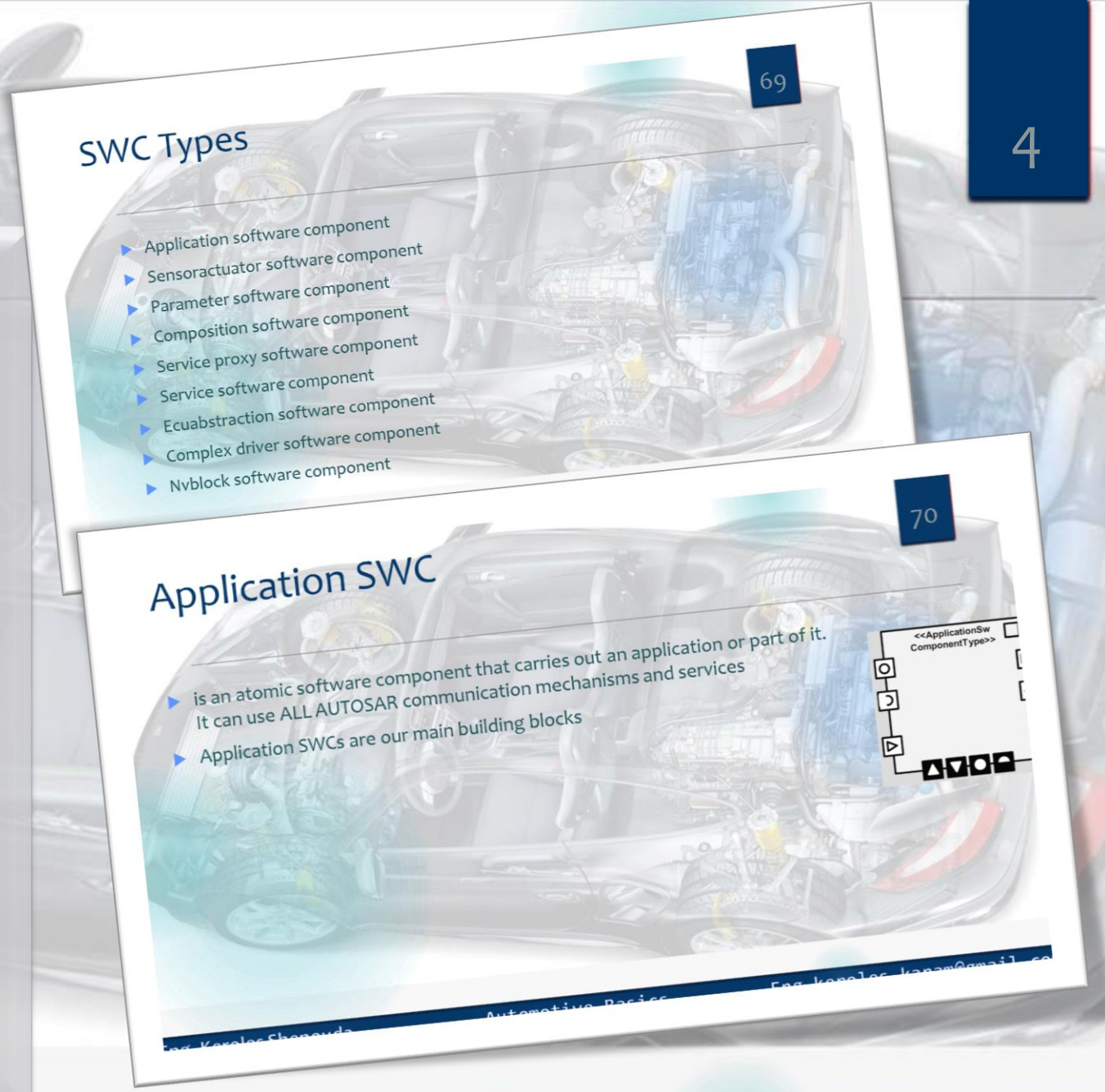
- Autosar Layered Architecture
- Example AUTOSAR System : Lighting System
  - CAN Stack example
- Autosar Interfaces:
  - Client-Server Communication:
  - Sender-Receiver Communicatio
- AUTOSAR TOOLS





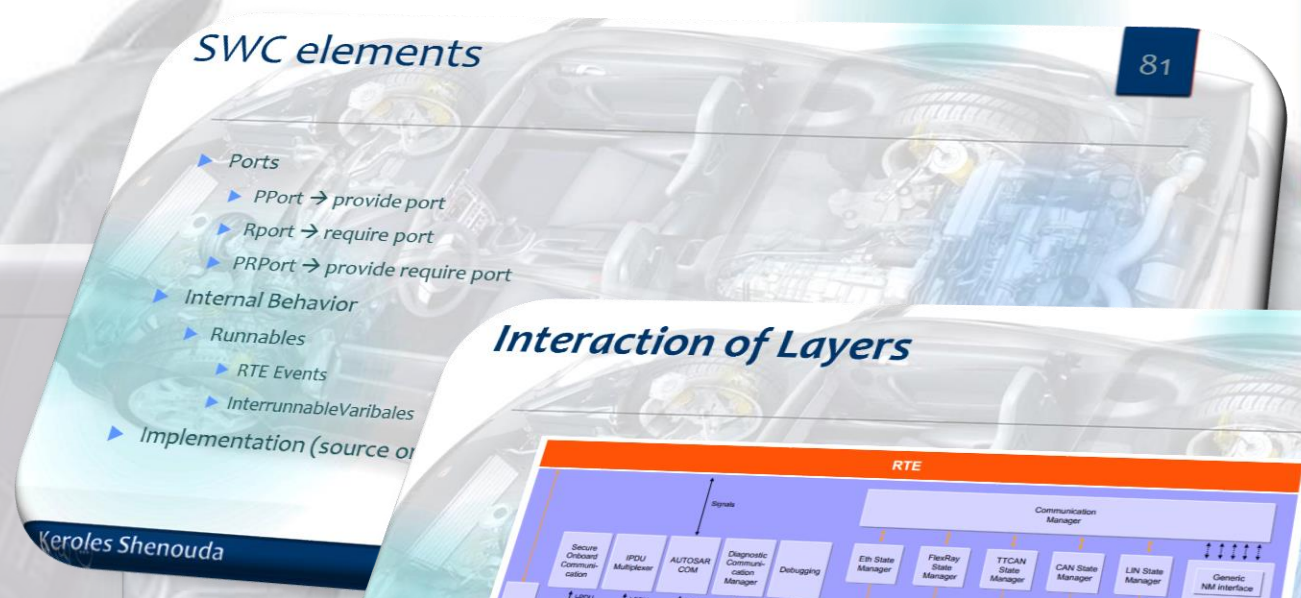
Index 3

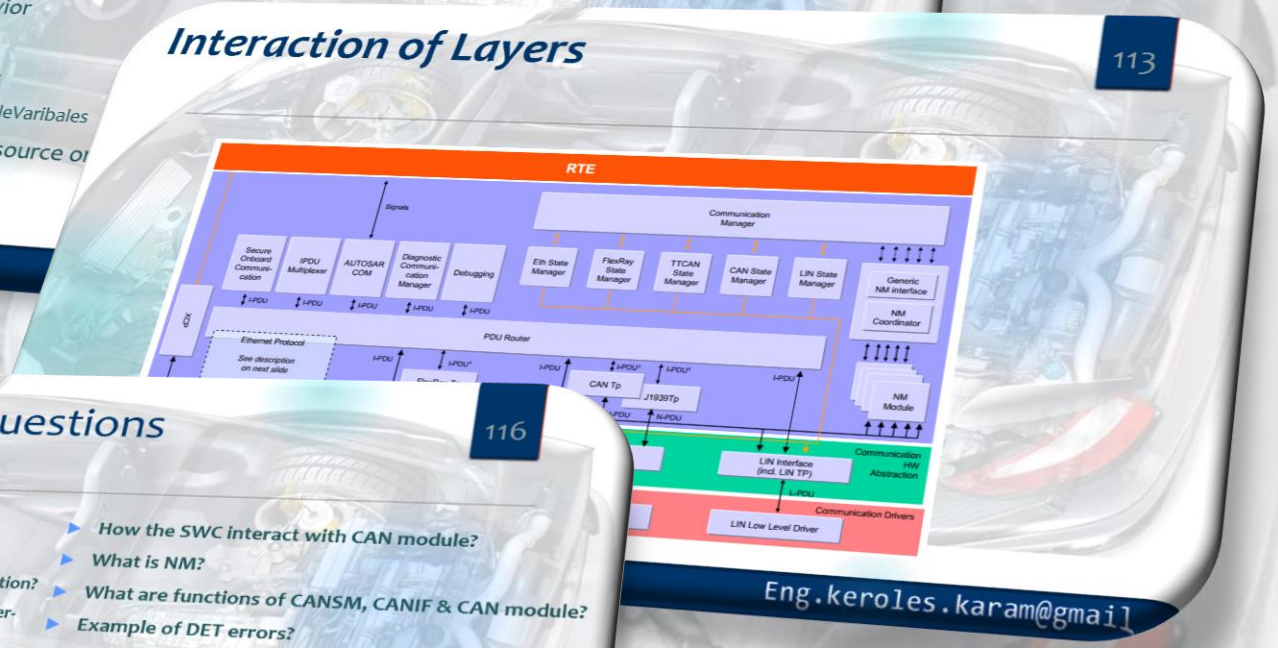
- AUTOSAR Software Component
  - Application SWC
  - Sensoractuator software component
  - Parameter Software Component
  - Composition software components
  - Service proxy SWC
  - Service software component
  - ECU-abstraction software component
  - SWC Connectors
  - Example of access pattern to sensors and actuators:
  - Complex driver software component
  - NVBlock SWC

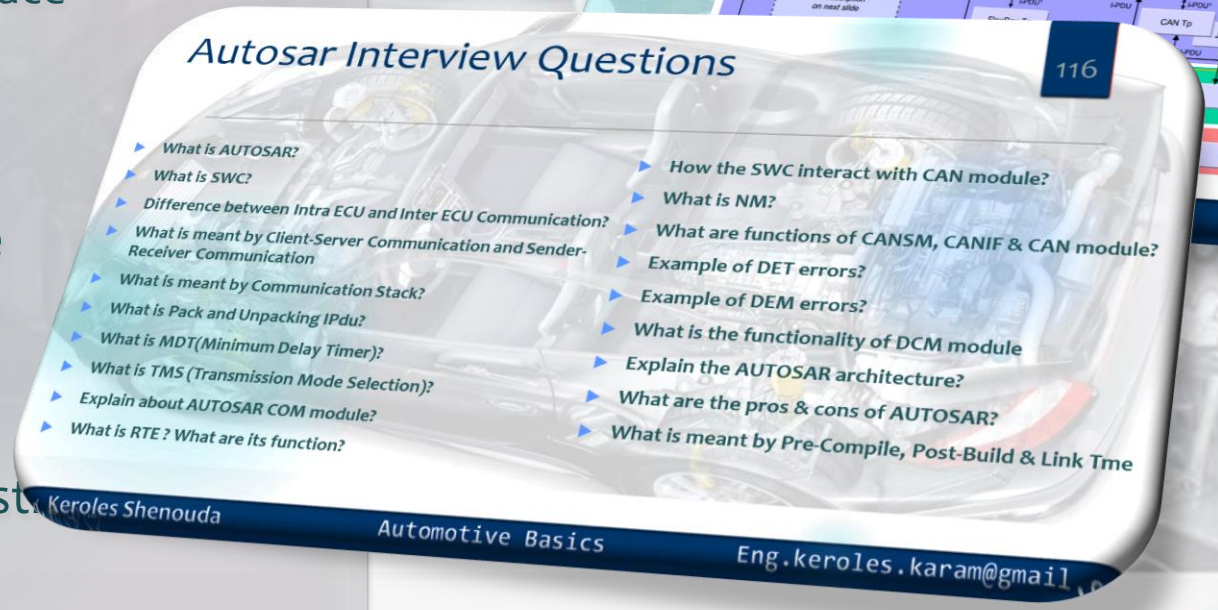


Index 4

- SWC elements
- SW Components and Runnables
- SWC Description and Elements
- Port Interfaces
  - SenderReceiverInterface
  - NvDataInterface
  - ParameterInterface
  - ModeSwitchInterface
  - ClientServerInterface
  - TriggerInterface
- Autosar Interview Quest

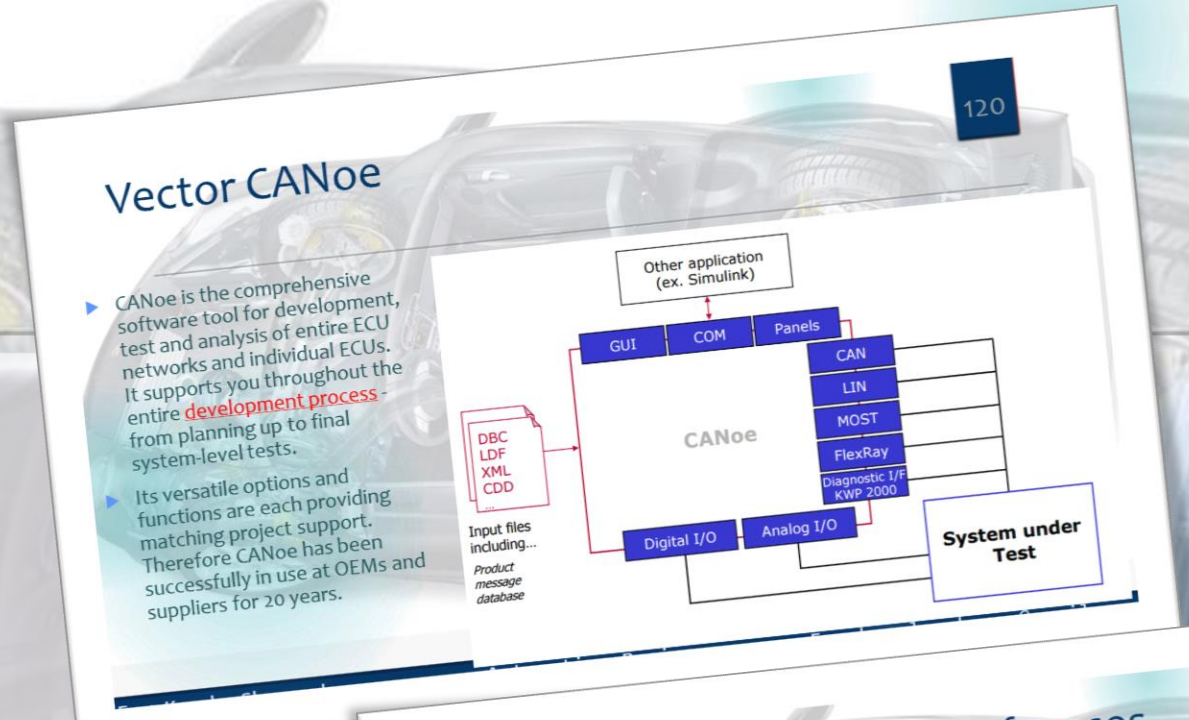






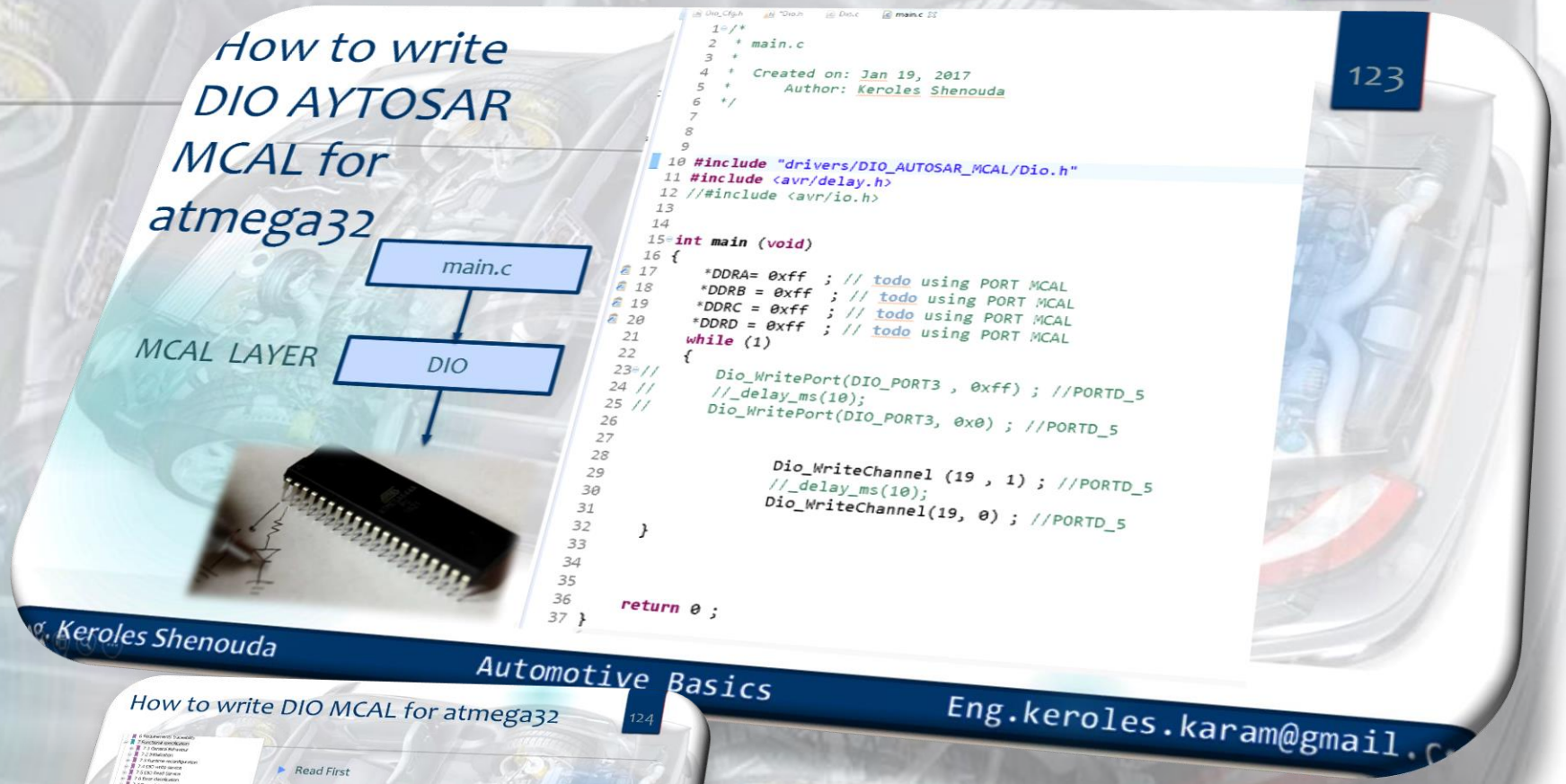
Index 5

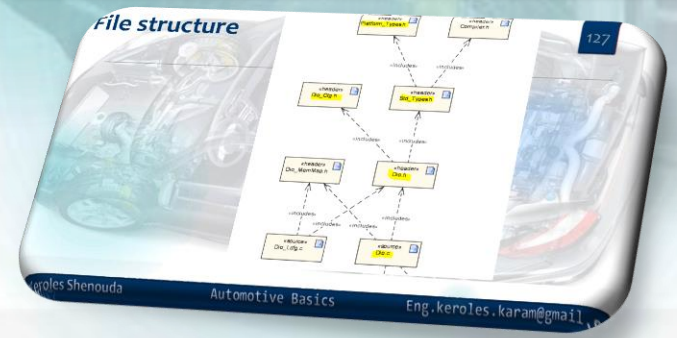
- Automotive Modeling
- ECU Testing by Canoe
- Vector CANoe
- Autosar Documentations References



Index 6

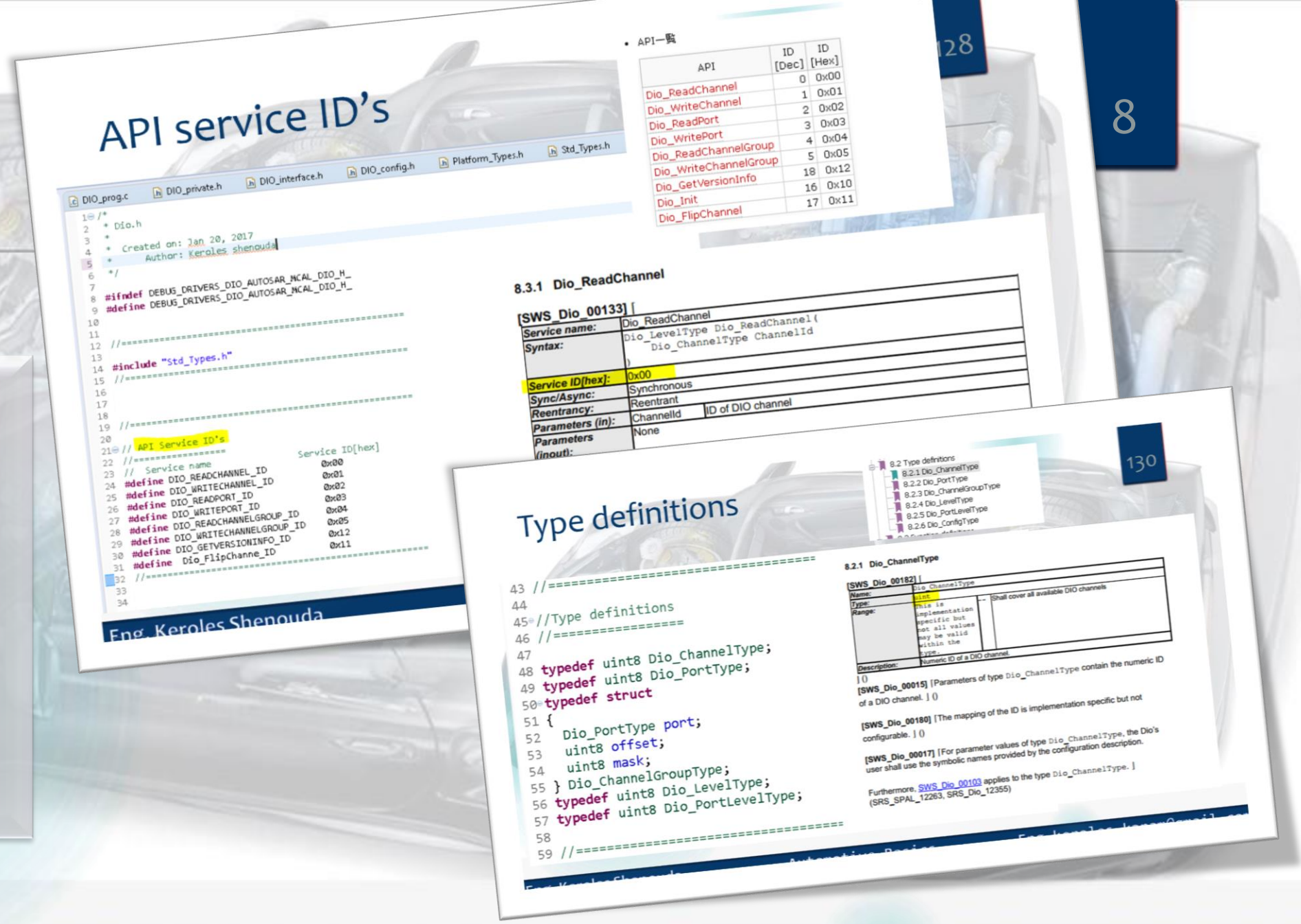
- How to write DIO AYTOSAR MCAL for atmega32
  - **DIO Driver Structure and Integration**
  - **Dependencies to other modules**
  - **File structure**





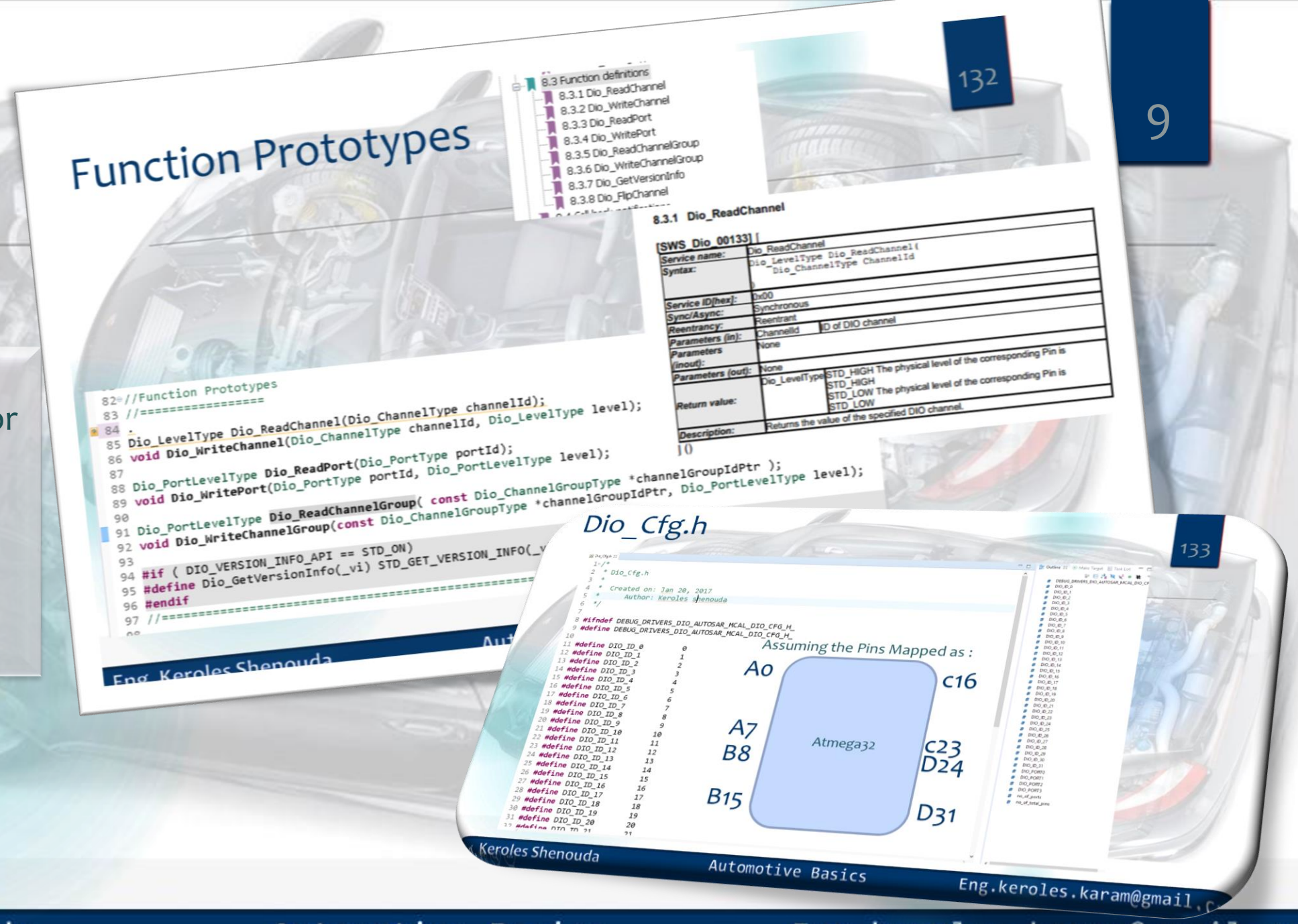
Index 7

- How to write DIO AYTOSAR MCAL for atmega32
  - API service ID's
  - **Error classification**
  - Type definitions
  - Version Number
  - Function Prototypes



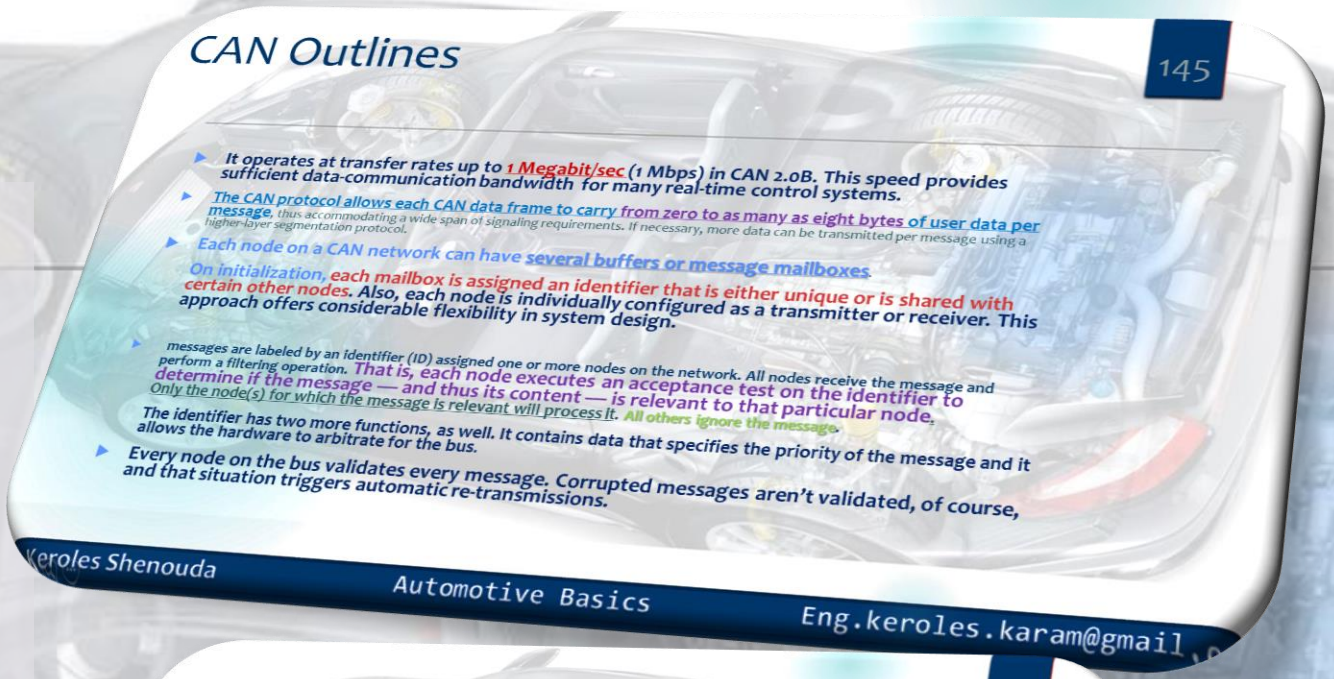
Index 8

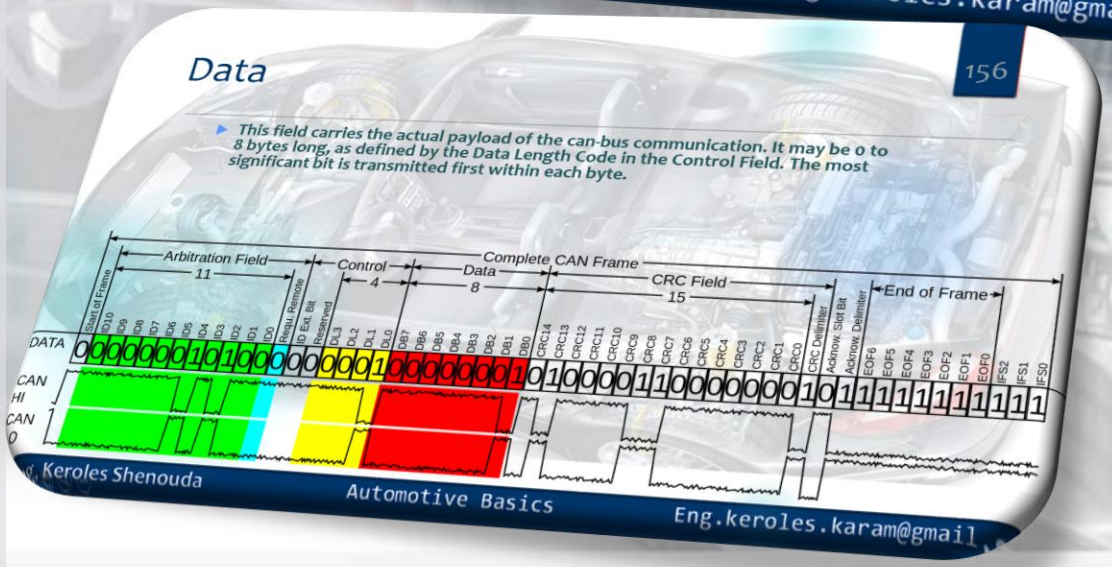
- How to write DIO AYTOSAR MCAL for atmega32
  - Dio_Cfg.h
  - APIs Code



Index 9

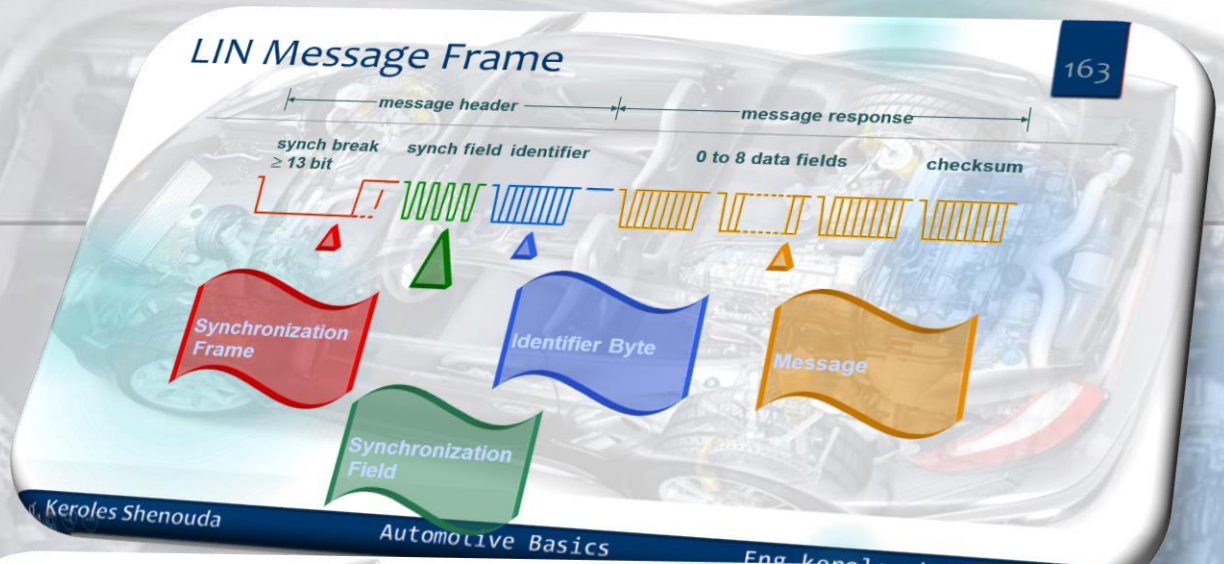
- What is CAN ?
- CAN-Leading Choice for Embedded Networking
- CAN Outlines
- CAN and the 7-layer model
- Data Flow in CAN
- Data Frame
- CAN and EMI
- CAN Baud Rate vs. Bus Length
- Error Detection in CAN
- Physical Layer

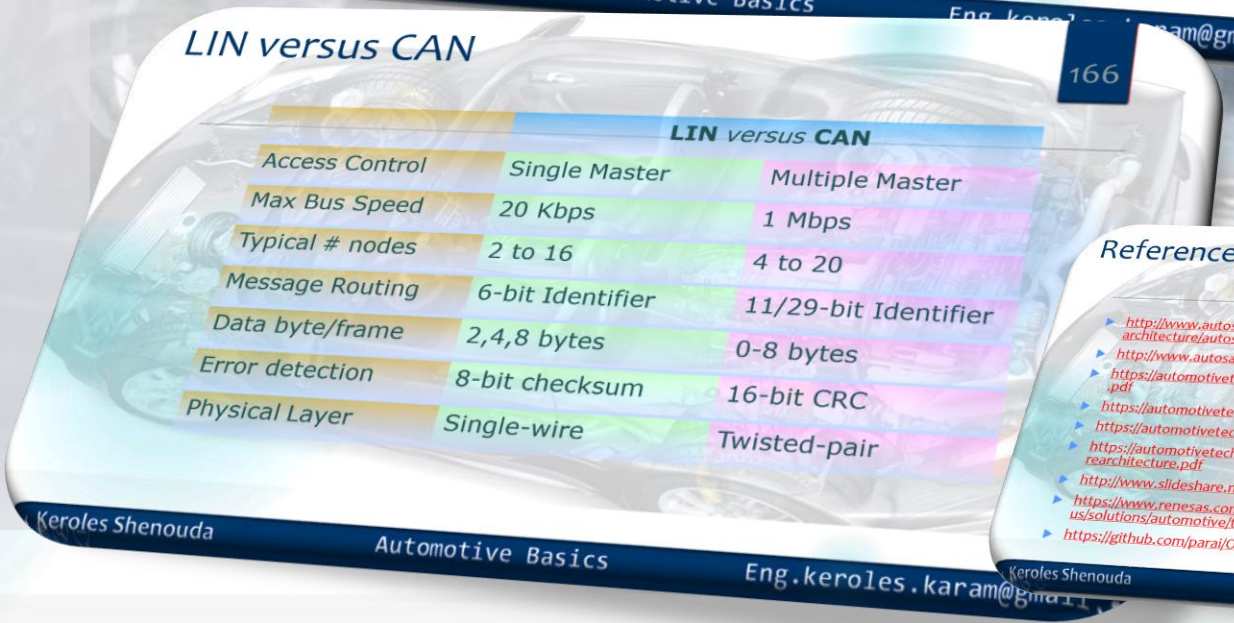




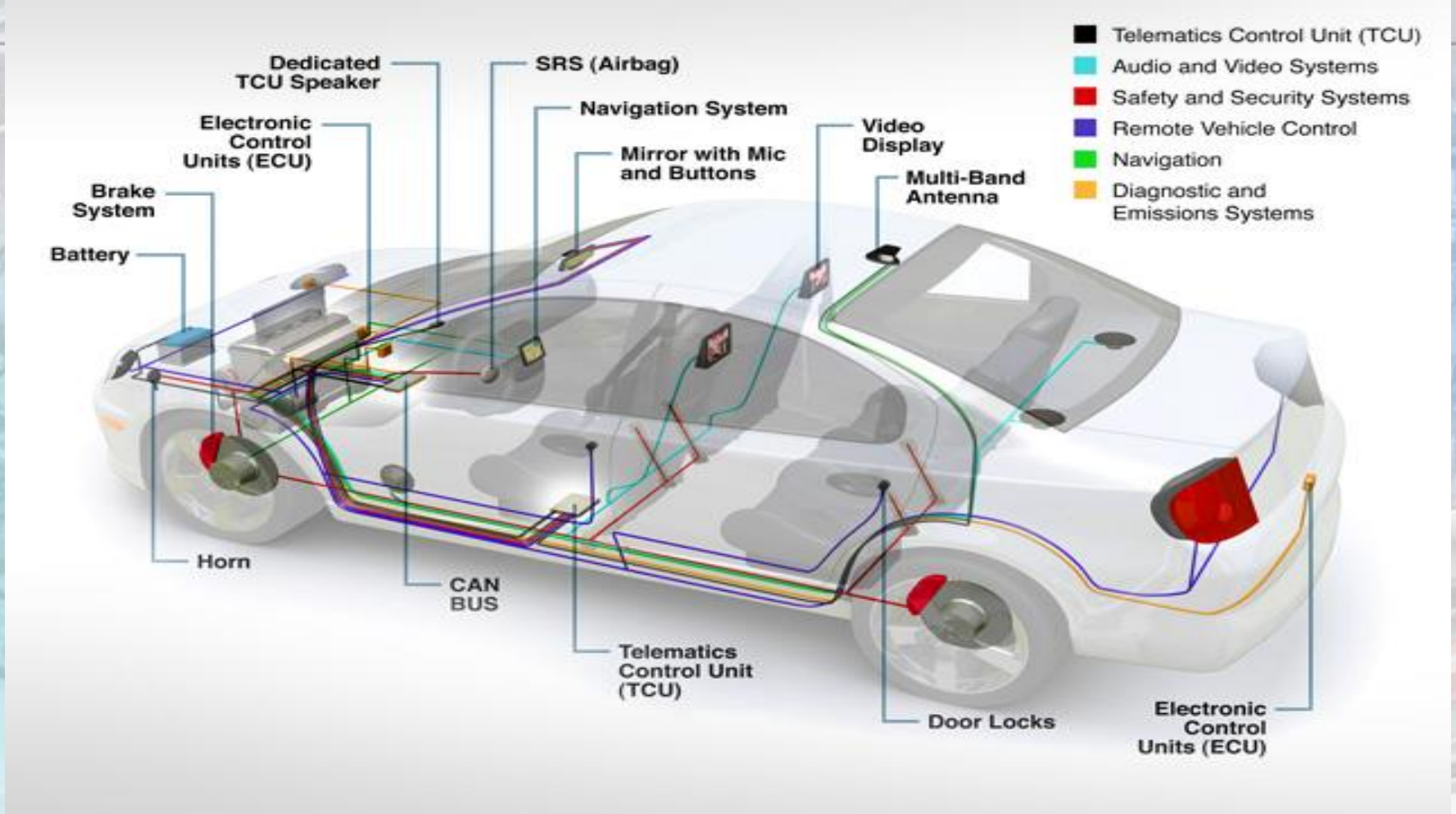
Index 10

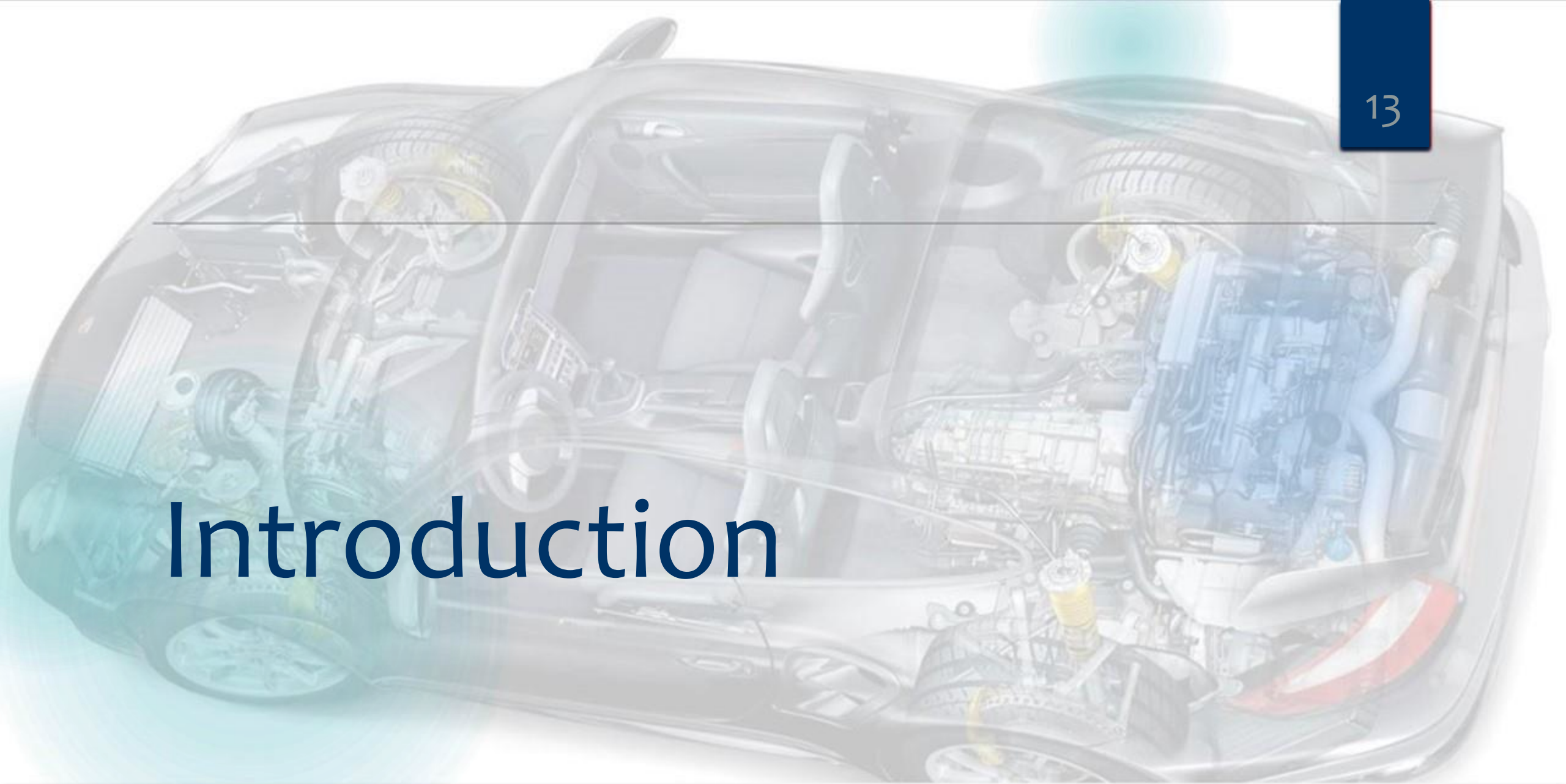
- What is LIN ?
  - Features & Benefits of LIN
  - Typical LIN Network
  - LIN Message Frame
  - Question
- LIN Vs CAN
- References





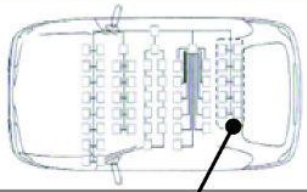
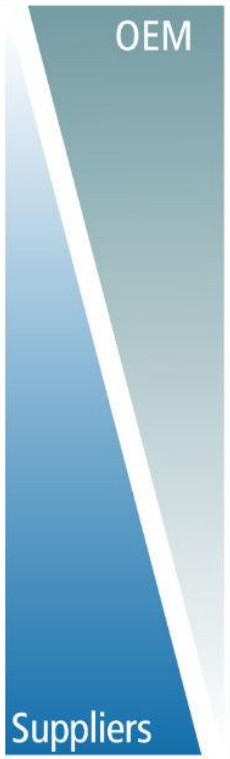
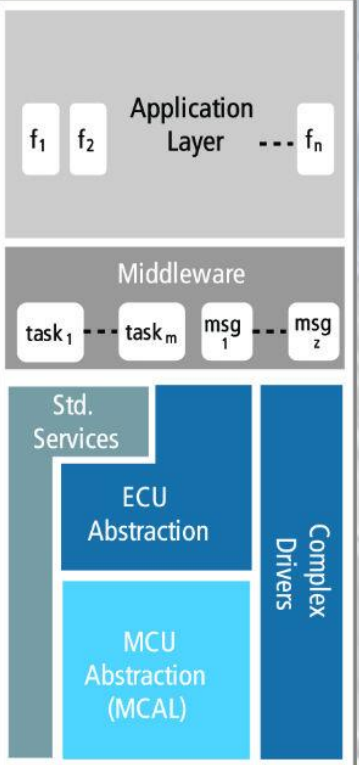
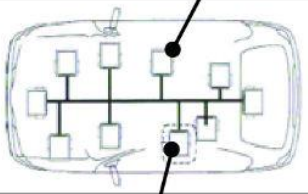
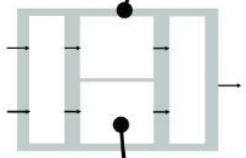
Embedded Automotive



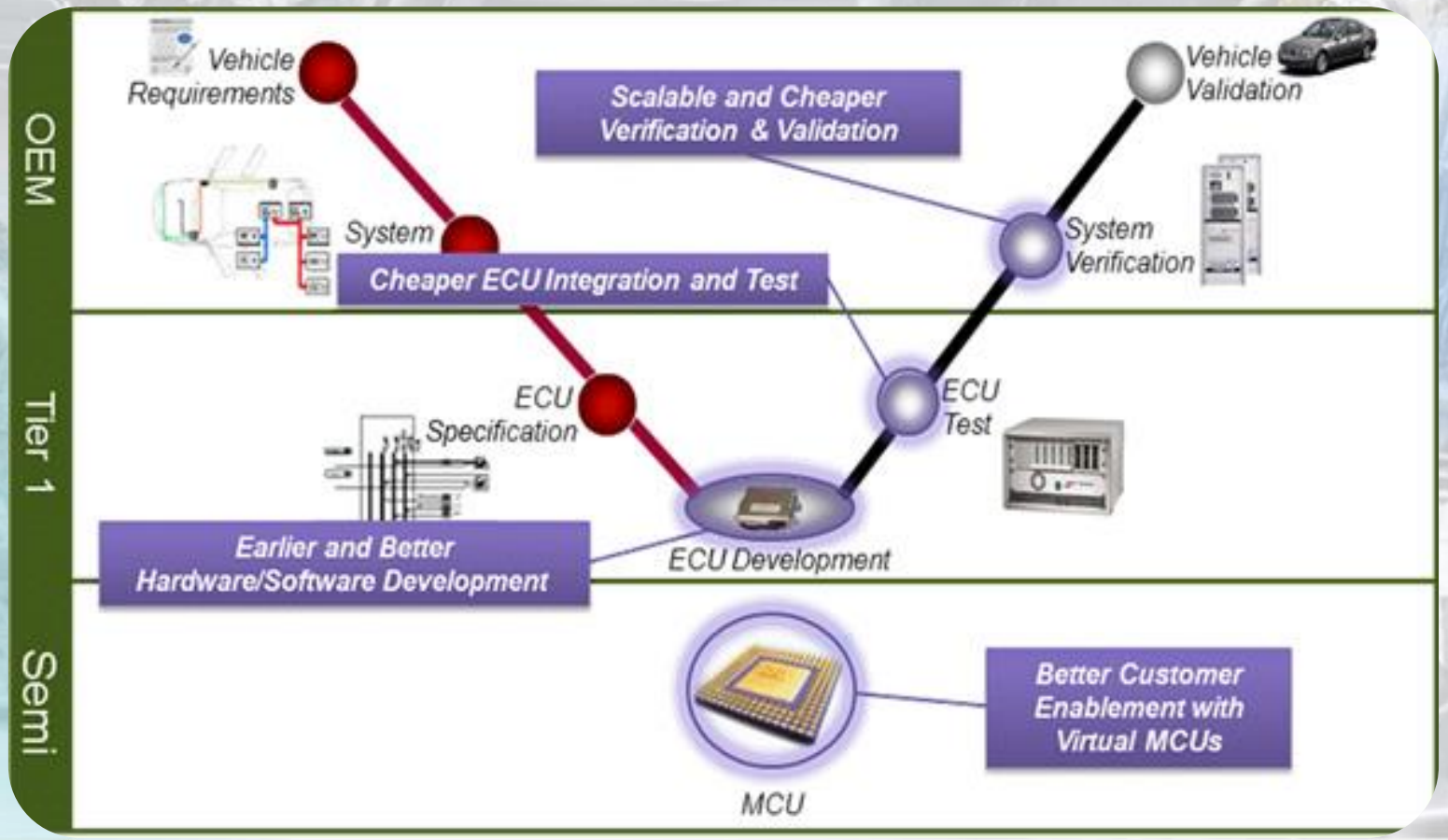


Introduction

Automotive industry

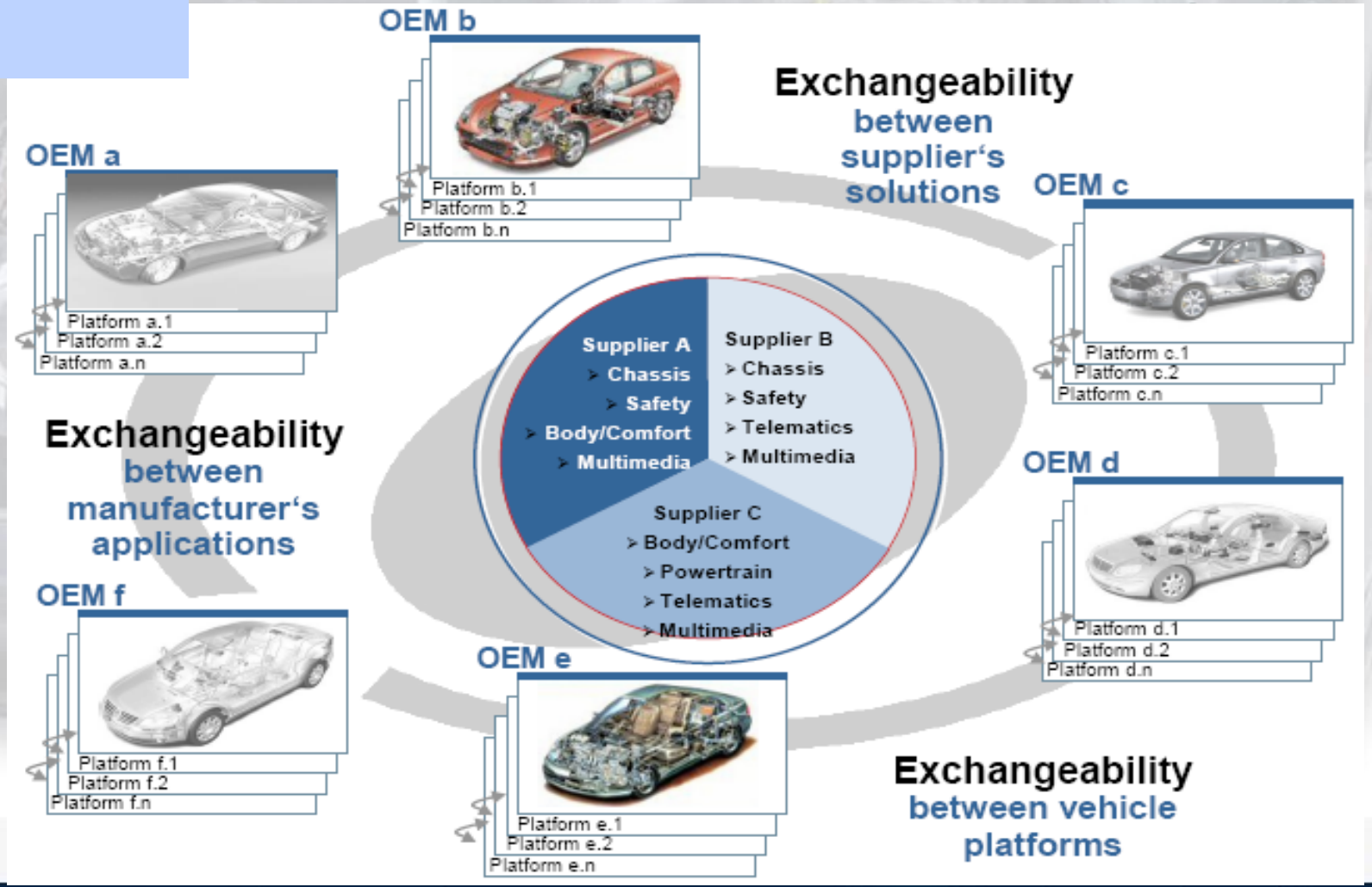
Domain	Domain	HW Scope	View	SW Scope
OEM	Vehicle			
OEM Tier 1	Subsystem			
Tier 1	ECU			
Tier 2	MCU			

Automotive Cycle



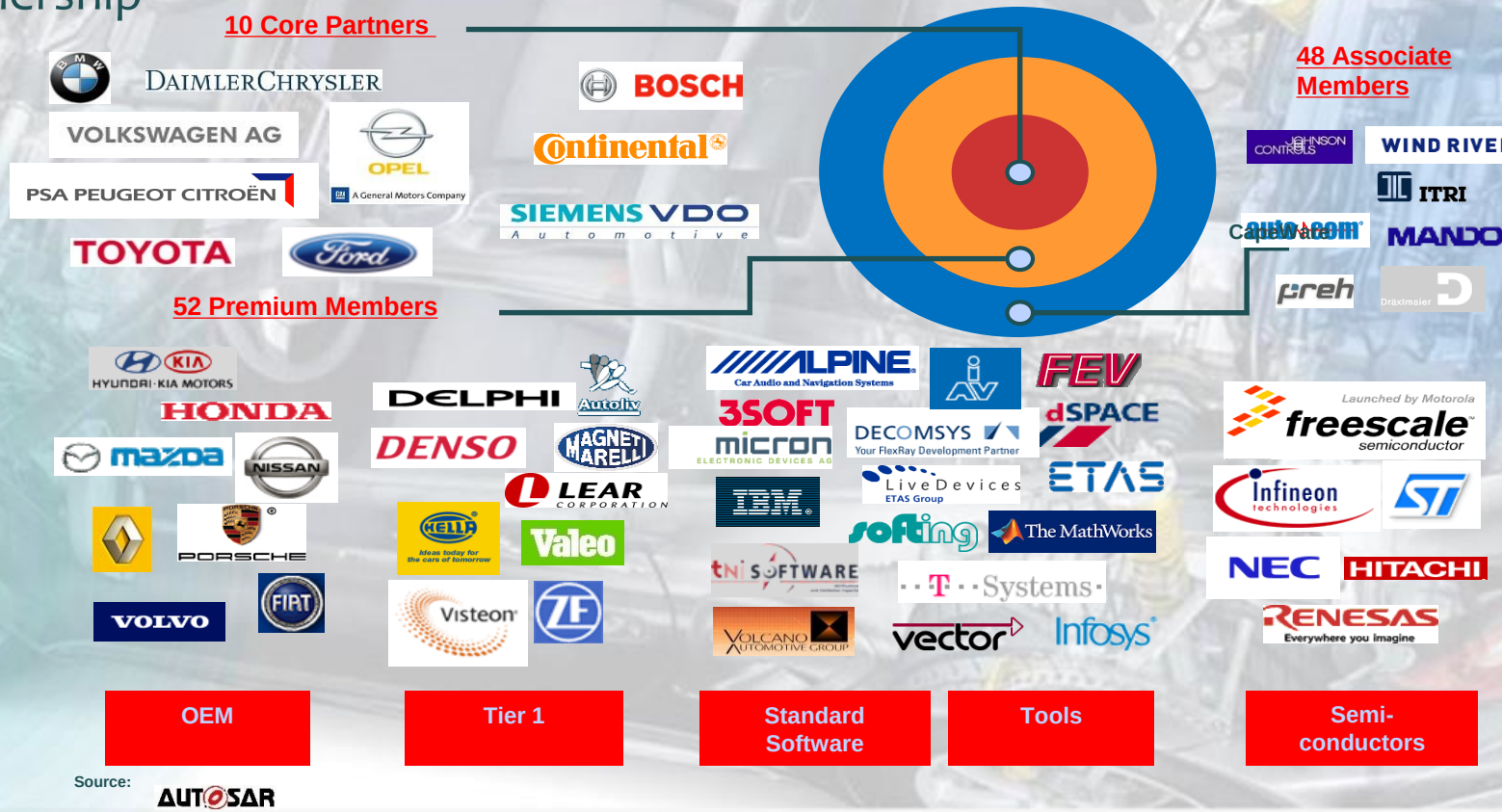
What is AUTOSAR

- AUTOSAR Project Objectives
  - AUTOSAR vision is an improved complexity management of highly integrated E/E architectures through an increased reuse of SW modules between OEM and suppliers.



What is AUTOSAR

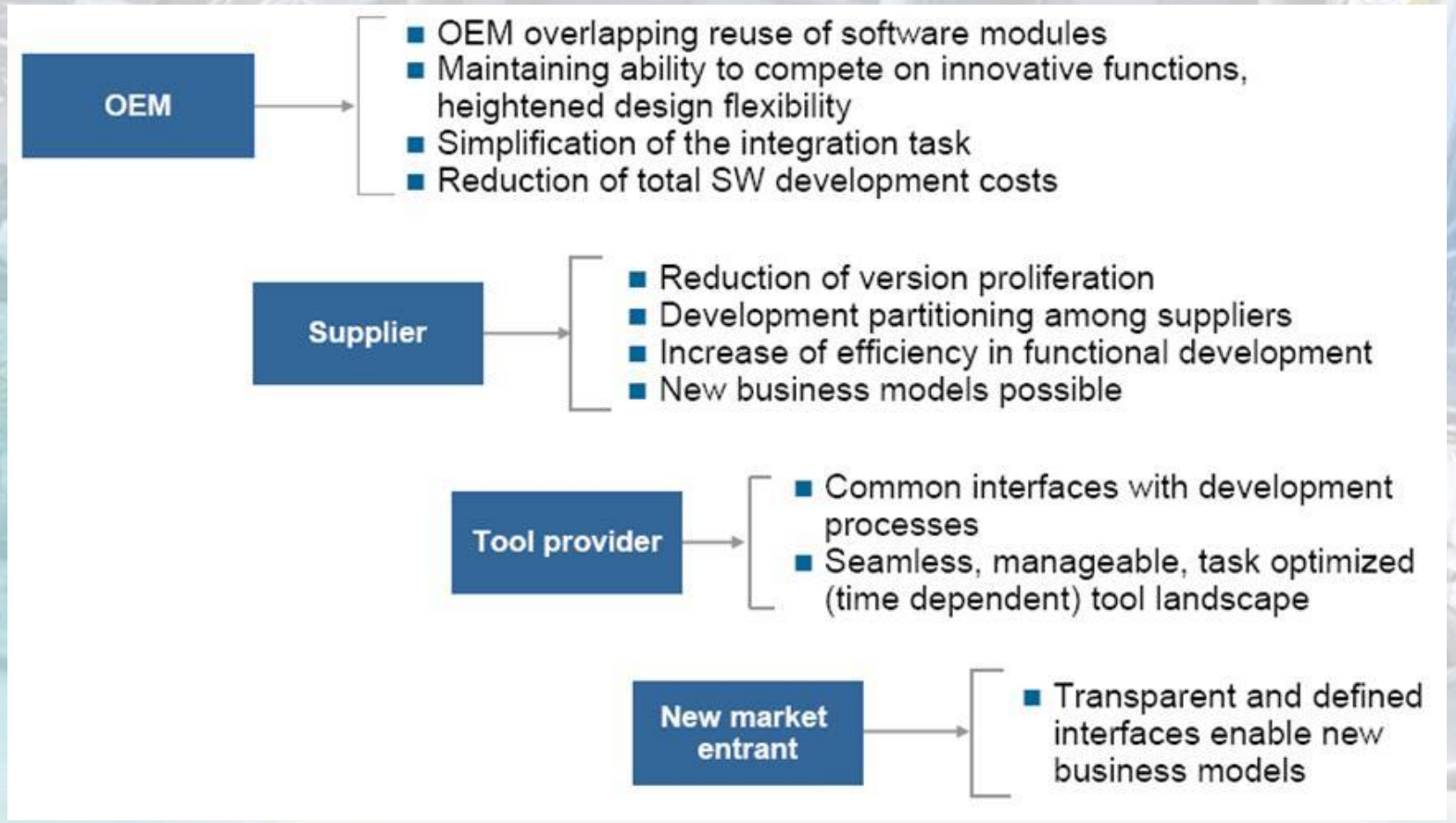
- AUTOSAR partnership



What is Autosar?

- **AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture**, jointly developed by automobile manufacturers, suppliers and tool developers. The AUTOSAR-standard enables the use of a component based software design model for the design of a vehicular system. The design model uses application software components which are **linked through an abstract component**, named **the virtual function bus.**
- **The *application software components* are the smallest pieces of application software that still have a certain functionality**. The software of an application can then be composed by using different application software-components. **Standardized interfaces for all the application software components necessary to build the different automotive applications are specified in the AUTOSAR-standards**. By only defining the interfaces, there is still freedom in the way of obtaining the functionality.
- **The virtual function bus connects the different software components** in the design model. This abstract component interconnects the different application software components and handles the information exchange between them. The virtual function bus is the conceptualization of all hardware and system services offered by the vehicular system. **This makes it possible for the designers to focus on the application instead of the infrastructure software.**

Benefits of Autosar:



How are vehicle functions implemented today?

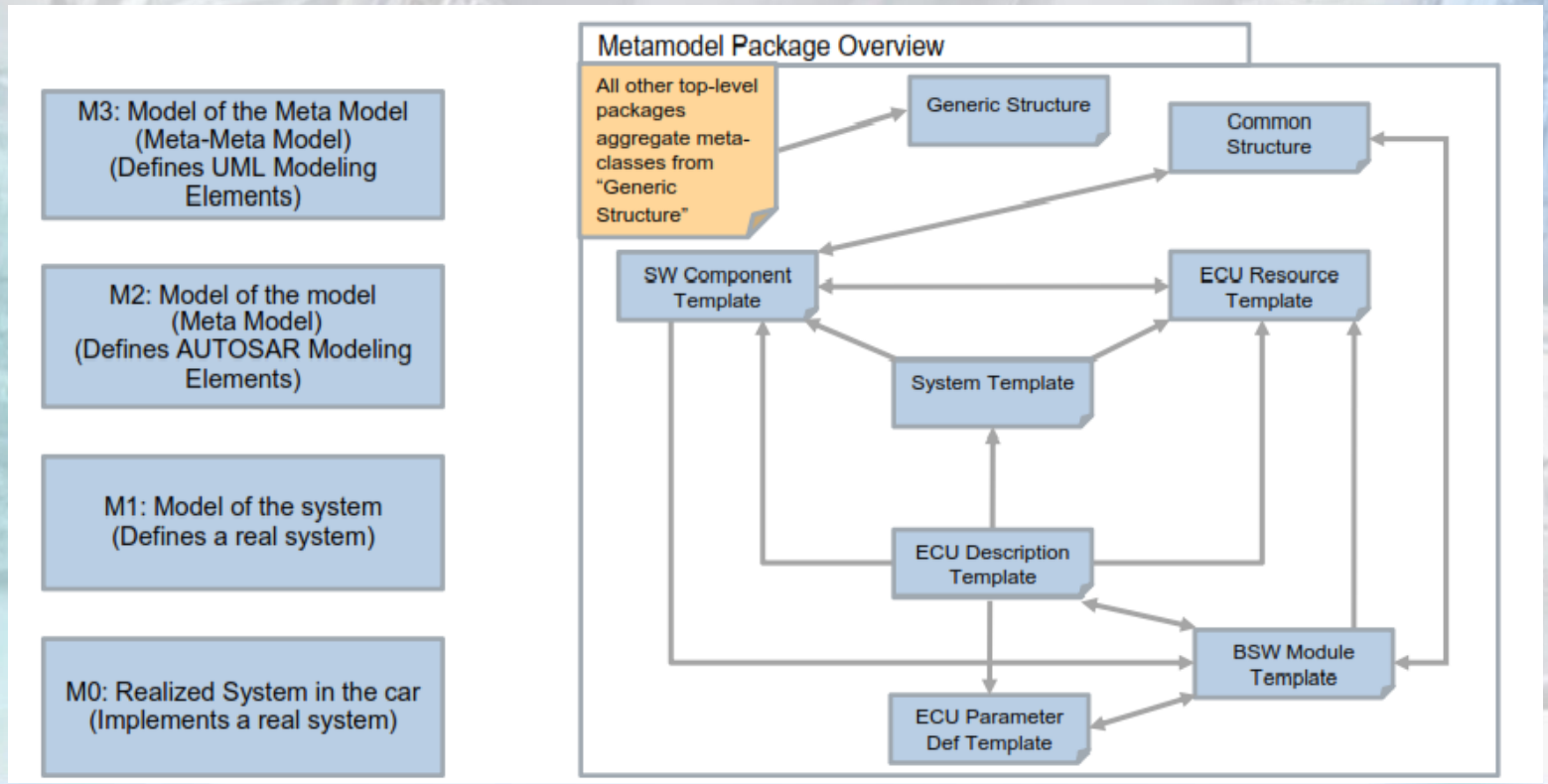
- Each function has it´s own system although they may communicate through a bus
- Hardware and software are tightly connected
- Each function has it´s own microcontroller
- The number of ECU´s (Electronic Control Unit) are growning fast
- The same vendor supplies both the hardware and the software
- There are no alternative software suppliers

What will Autosar give?

- A **standard** platform for vehicle software
- An **OS** with basic functions and interface to software
- **Functionality** is supplied as **software components**
- An with **basic functions and interface** to software
- These components are **hardware independent**
- No applications of its own
- The same interface **for all basic software**
- The software can be **reused**
- **More than one supplier** can compete with their software

AUTOSAR Meta Model

- **AUTOSAR Meta Model** is the backbone of the AUTOSAR architecture definition contains complete specification, how to model AUTOSAR system.

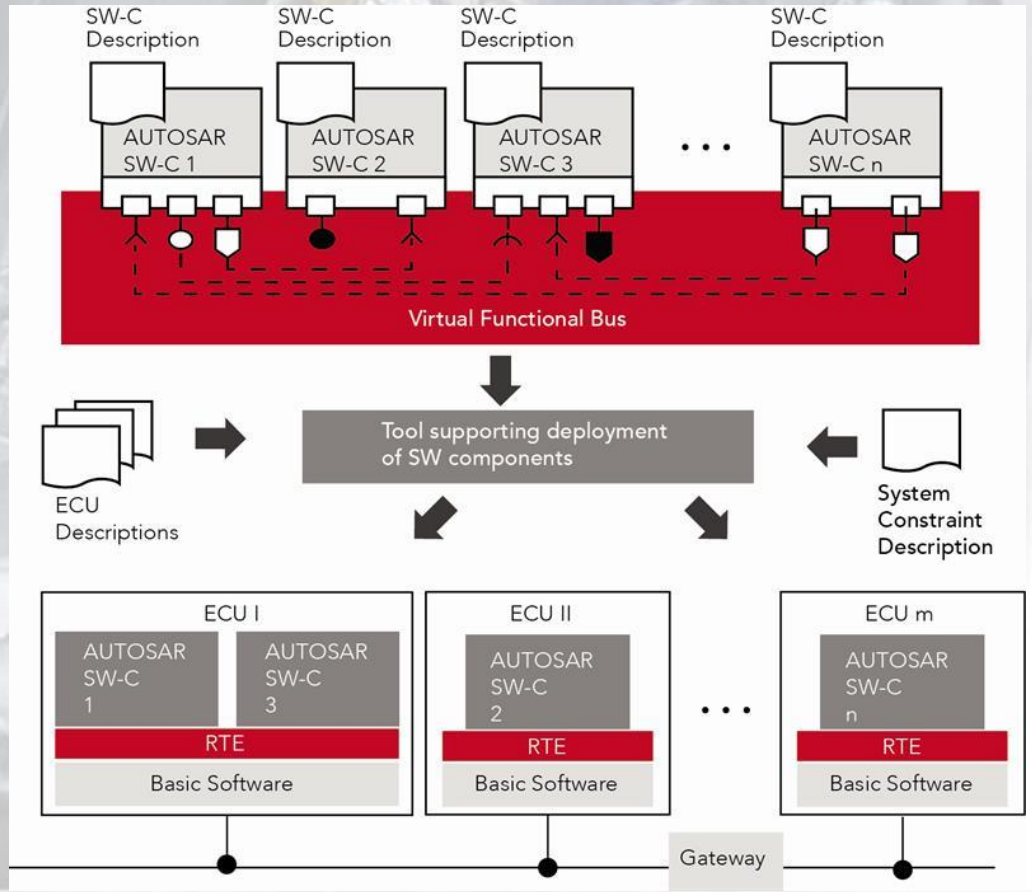


What is Autosar? Cont.

- By using the virtual function bus, the application software components do not need to know with which other application software components they communicate. ***The software components*** **give their output to the virtual function bus, which guides the information to the input ports of the software components that need that information.** This is possible due to the standardized interfaces of the software components which specifies the input and output ports as well as the format of data exchange.
- This approach makes it possible to validate the interaction of all components and interfaces before software implementation. **This is also a fast way to make changes in the system design and check whether the system will still function.**
- **The AUTOSAR-project created a methodology that can be used to create the E/E system architecture starting from the design-model. This approach uses 4 steps:**

What is Autosar? Cont.

- Step 1: Input Descriptions
- Step 2: System Configuration
- Step 3: ECU-configuration
- Step 4: Generation of Software Executables



Step 1: Input Descriptions

- The input description step contains three descriptions:
  - **Software Components**: This description is independent of the actual implementation of the software component. Among the necessary data to be specified are the interfaces and the hardware requirements.
  - **System**: The system topology (interconnections between ECUs) need to be specified together with the available data busses, used protocols, function clustering and communication matrix and attributes (e.g. data rates, timing/latency, …).
  - **Hardware**: The available hardware (processors, sensors, actuators, …) needs to be specified together with the signal processing methods and programming capabilities

Step 2: System Configuration

- This step distributes the software component descriptions to the different ECU. This is an iterative process where ECU-resources and system-constraints are taken into account. For example, there needs to be checked whether the necessary communication-speeds are met.

Step 3: ECU-configuration

- In this step, the Basic Software and the Run Time Environment of each electronic control unit (ECU) is configured. This is based on the dedication of the application software components to each ECU.

Step 4: Generation of Software Executables

- Based on the configuration of the previous step, **the software executables are generated**. For this step, it's necessary to specify the **implementation** of each software component.
- This methodology is automated by using tool-chains. All subsequent methodology steps up to the generation of executables are supported by defining exchange formats (using **XML**) and work methods for each step.
- **To support the Autosar-methodology, a metamodel is developed. This is a formal description of all methodology related information, modeled in UML. This leads to the following benefits:**
  1. **The structure of the information can be clearly visualized**
  2. **The consistency of the information is guaranteed**
  3. **Using XML, a data exchange format can be generated automatically out of the meta-model and be used as input for the methodology.**
  4. **Easy maintenance of the entire vehicular system**

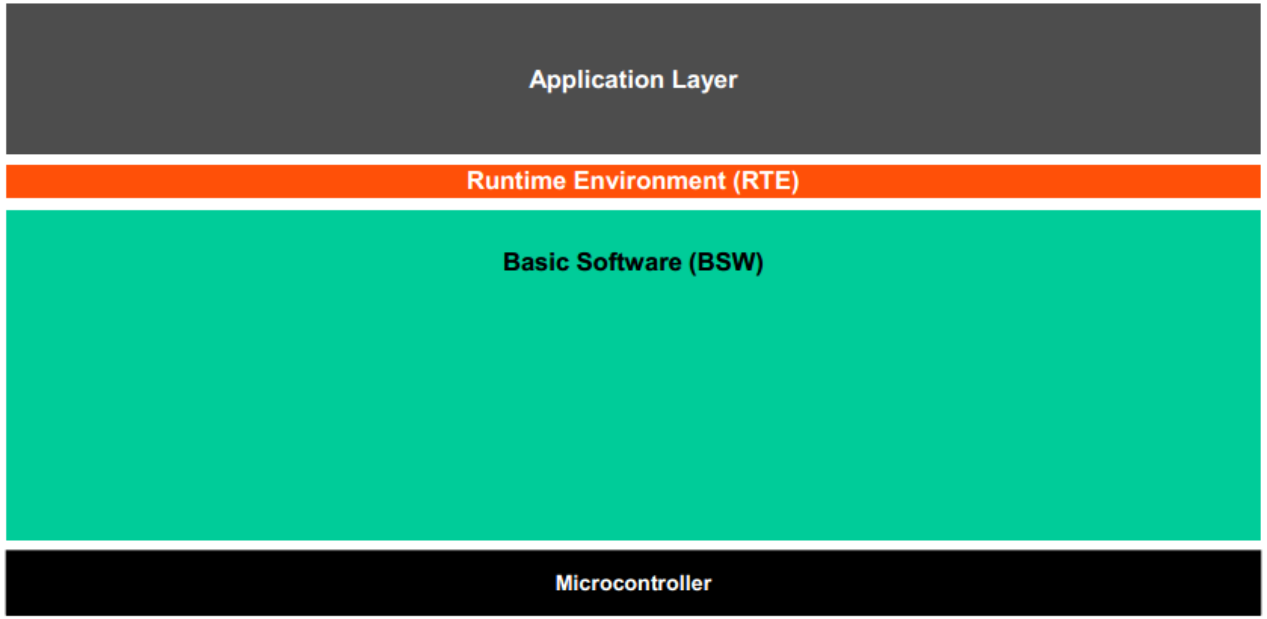
Step 4: Generation of Software Executables (Cont.)

- There are four types of membership for AUTOSAR:
  - Core (founding) members
  - Premium members
  - Associate members
  - Development members
- Core membership only is available for leading car manufacturers and Tier1; the other types of membership are open to other companies as well.
- Core members include the Toyata Motor Corporation, Volkswagen , BMW Group, Daimler AG, Ford Motor Company, Opel , and automotive suppliers Bosch, Continental AG and Siemens VDO (now Continental AG).

Autosar Layered Architecture

Architecture – Overview of Software Layers
Top view

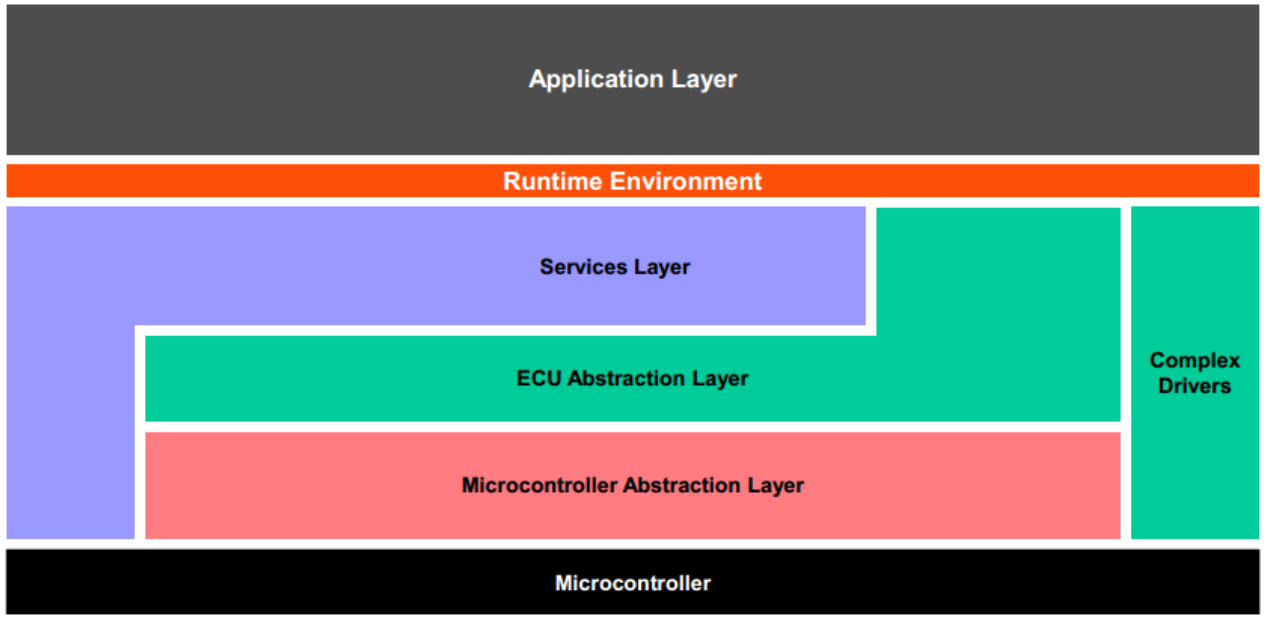
The AUTOSAR Architecture distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller.



Autosar Layered Architecture (Cont.)

Architecture – Overview of Software Layers
Coarse view

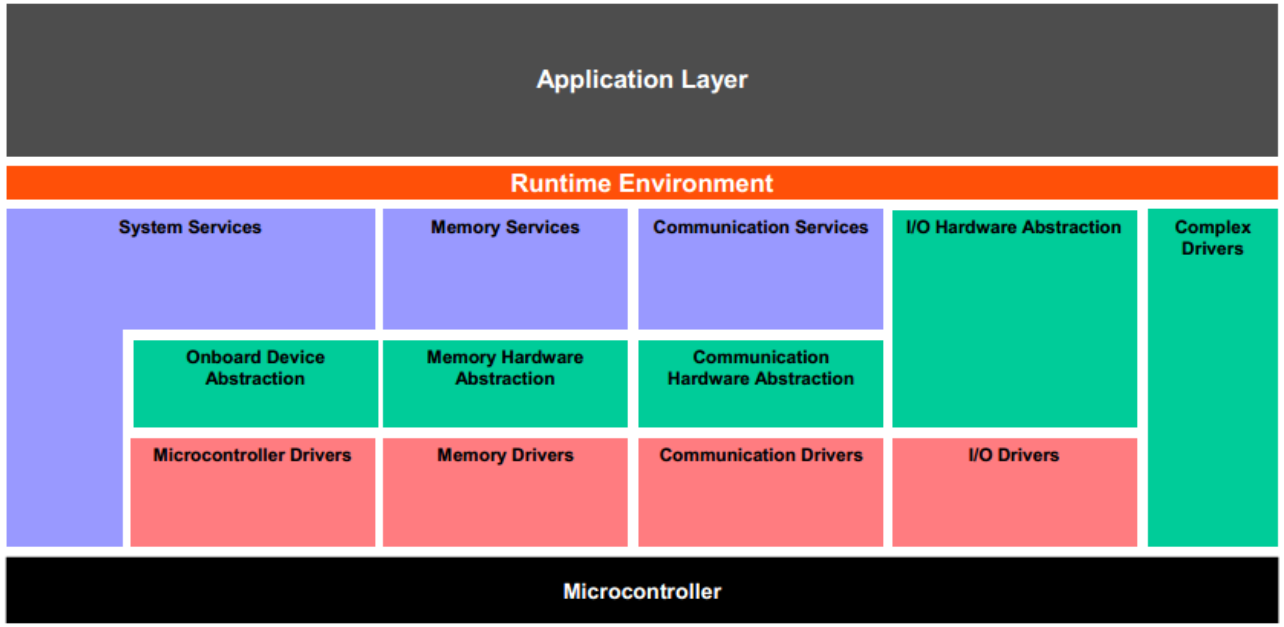
The AUTOSAR Basic Software is further divided in the layers: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers.



Autosar Layered Architecture (Cont.)

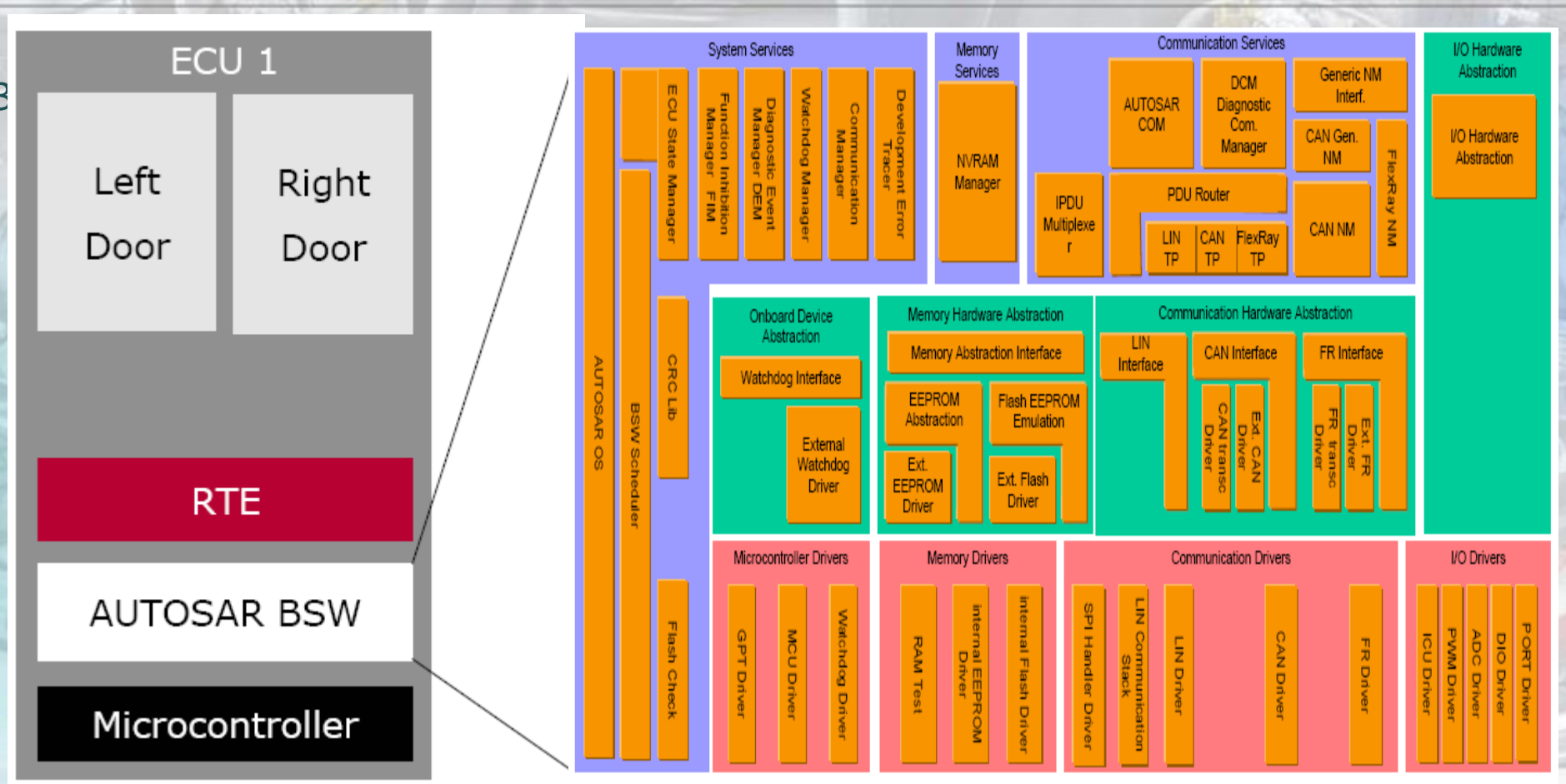
Architecture – Overview of Software Layers
Detailed view

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



Example AUTOSAR System : Lighting System

- B

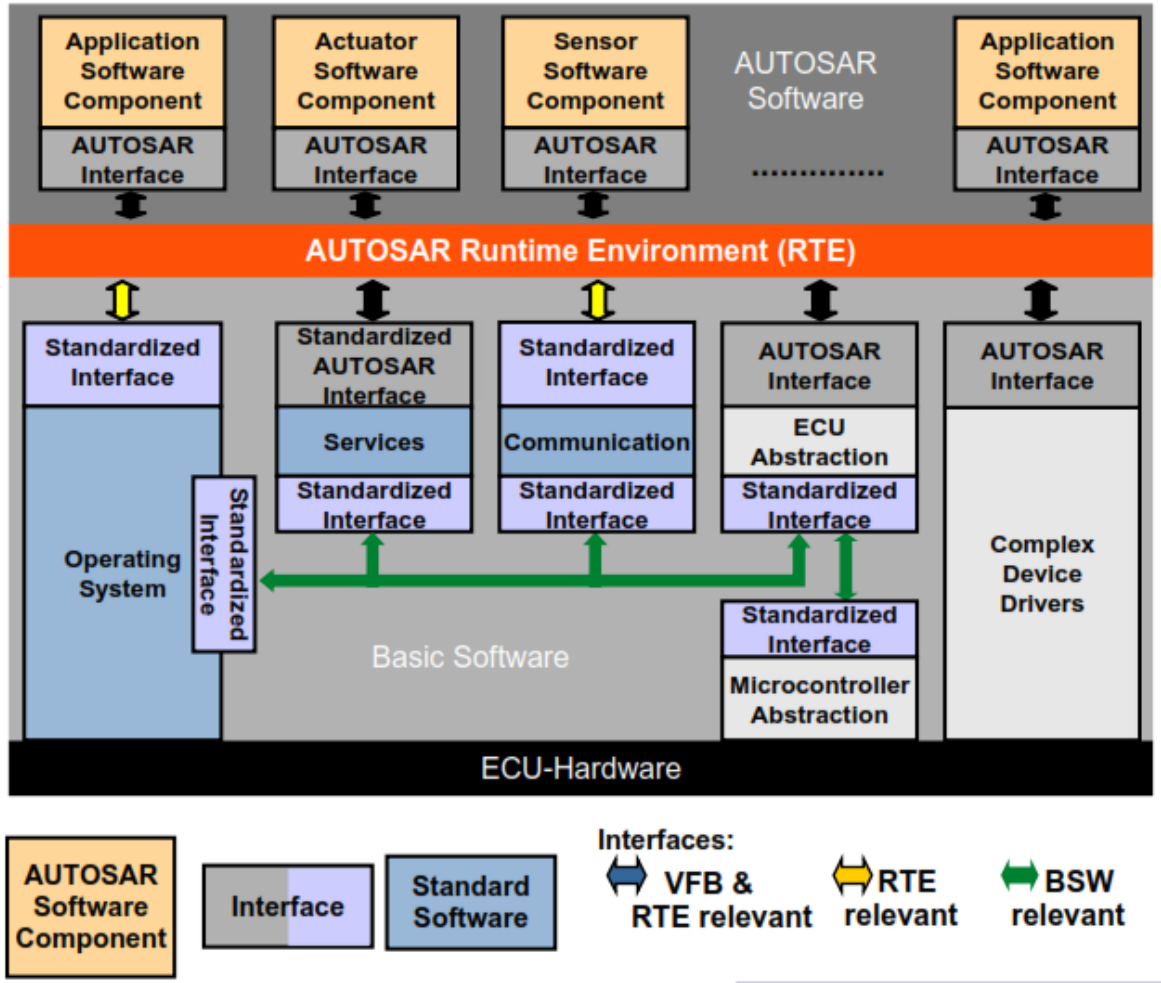


Autosar Layered Architecture (Cont.)

Classification Of Interfaces:
There are three different types of interfaces in Autosar Layered Architecture.

Standardized Autosar Interfaces:
•A Standardized AUTOSAR Interface is an AUTOSAR Interface standardized within the AUTOSAR project.

Standardized Interfaces:
•A software interface is called Standardized Interface if a concrete standardized API exists (e.g. OSEK COM Interface Com_ReceiveSignal & Com_TransmitSignal which are called by RTE module)



www.autosar.org

Apps Bookmarks Getting Started my life email mentor graphics vpn courses projects tricks community

AUTOSAR
Enabling Innovation

Type your search here

About Partners Specifications Documents User Groups Events & Publications Media

Release 4.2
Release 4.1
Release 4.0
Release 3.2
Release 3.1
Release 3.0
Release 2.0
Acceptance Tests for Classic Platform Release 1.1
Acceptance Tests for Classic Platform Release 1.0

Welcome to the AUTOSAR

AUTOSAR (AUTomotive Open System ARc ship of vehicle manufacturers, suppliers and other or and software industry.

AUTOSAR

- Paves the way for innovative electronic fety and environmental friendliness
- Is a strong global partnership that crea ndards, compete on implementation"
- Is a key enabling technology to manage growing electrical/electronic complexity. It aims to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality
- Facilitates the exchange and update of software and hardware over the service life of the vehicle

News

AUTOSAR Administration moves to a new office.

From 1st Dec. 2015
please use the new address.

read more

Publications

Papers & Presentations

10 Years AUTOSAR
AUTOSAR published a special issue ATZextra

read more

www.autosar.org/specifications/release-42/software-architecture/general/

Apps | Bookmarks | Getting Started | my life | email | mentor graphics | vpn | courses | projects | tricks | community

About | Partners | Specifications | Documents | User Groups | Events & Publications | Media

Specifications

- **Release 4.2**
  - Main
  - Software Architecture
    - General
    - Communication Stack
    - System Services
    - Diagnostic Services
    - Memory Stack
    - Peripherals
    - Implementation Integration
    - RTE
    - Libraries
    - Safety and Security
  - Methodology and Templates
  - Application Interfaces
- **Release 4.1**
- **Release 4.0**
- **Release 3.2**
- **Release 3.1**
- **Release 3.0**
- **Release 2.0**
- **Acceptance Tests for Classic Platform Release 1.1**
- **Acceptance Tests for**

Home | Specifications | Release 4.2 | Software Architecture | General

General

Standard Specifications
2 files

- AUTOSAR SWS BSWGeneral
  1.37 MB
- AUTOSAR SWS StandardTypes
  658.69 KB

Auxiliary Material
10 files

- AUTOSAR EXP ApplicationLevelErrorHandling
  984.05 KB
- AUTOSAR EXP BSWDistributionGuide
  1.34 MB
- AUTOSAR EXP CDDDesignAndIntegrationGuideline
  390.64 KB
- AUTOSAR EXP ErrorDescription
  1.01 MB
- AUTOSAR EXP InterruptHandlingExplanation
  392.45 KB
- AUTOSAR EXP LayeredSoftwareArchitecture
  2.9 MB
- AUTOSAR MOD BSWUMLModel
  24.39 MB
- AUTOSAR SRS BSWGeneral
  1.38 MB
- AUTOSAR TR BSWModuleList
  107.22 KB
- AUTOSAR TR BSWUMLModelModelingGuide
  1.28 MB

www.autosar.org/specifications/release-42/software-architecture/communication-stack/

Apps | Bookmarks | Getting Started | my life | email | mentor graphics | vpn | courses | projects | tricks | community

About | Partners | Specifications | Documents | User Groups | Events & Publications | Media

Specifications

Release 4.2
- Main
- Software Architecture
  - General
  - Communication Stack
  - System Services
  - Diagnostic Services
  - Memory Stack
  - Peripherals
  - Implementation Integration
  - RTE
  - Libraries
  - Safety and Security
- Methodology and Templates
- Application Interfaces

Release 4.1

Release 4.0

Release 3.2

Release 3.1

Release 3.0

Release 2.0

Acceptance Tests for Classic Platform Release 1.1

Home | Specifications | Release 4.2 | Software Architecture | Communication Stack

Communication Stack

Standard Specifications
46 files

- AUTOSAR SWS CANDriver 1.88 MB
- AUTOSAR SWS CANInterface 1.76 MB
- AUTOSAR SWS CANNetworkManagement 1.87 MB
- AUTOSAR SWS CANStateManager 1.54 MB
- AUTOSAR SWS CANTransceiverDriver 1.51 MB
- AUTOSAR SWS CANTransportLayer 1.84 MB
- AUTOSAR SWS COM 2.95 MB
- AUTOSAR SWS COMManager 2.12 MB
- AUTOSAR SWS CommunicationStackTypes 650.47 KB
- AUTOSAR SWS EthernetDriver 1.24 MB
- AUTOSAR SWS EthernetInterface 1.46 MB
- AUTOSAR SWS EthernetStateManager 1.05 MB
- AUTOSAR SWS EthernetTransceiverDriver 1.05 MB
- AUTOSAR SWS FlexRayARTransportLayer 1.75 MB

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.

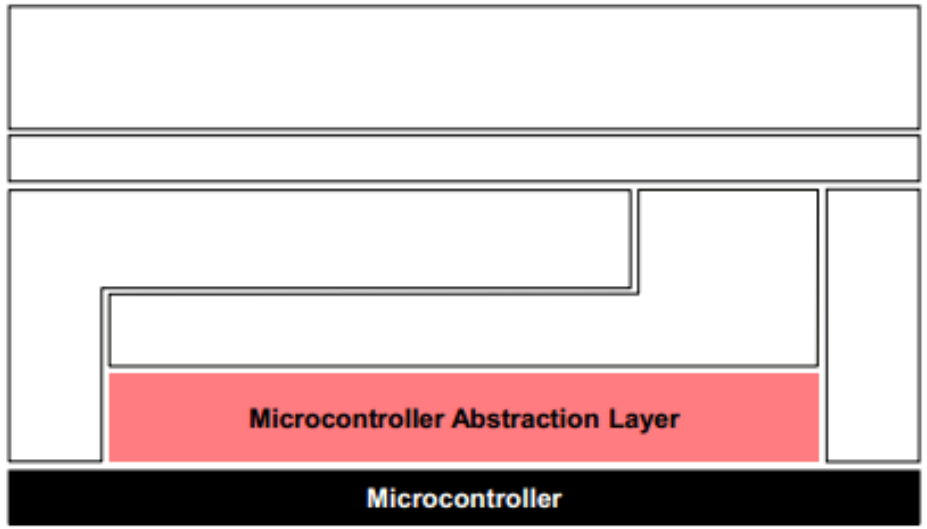
Task

Make higher software layers independent of µC

Properties

Implementation: µC dependent

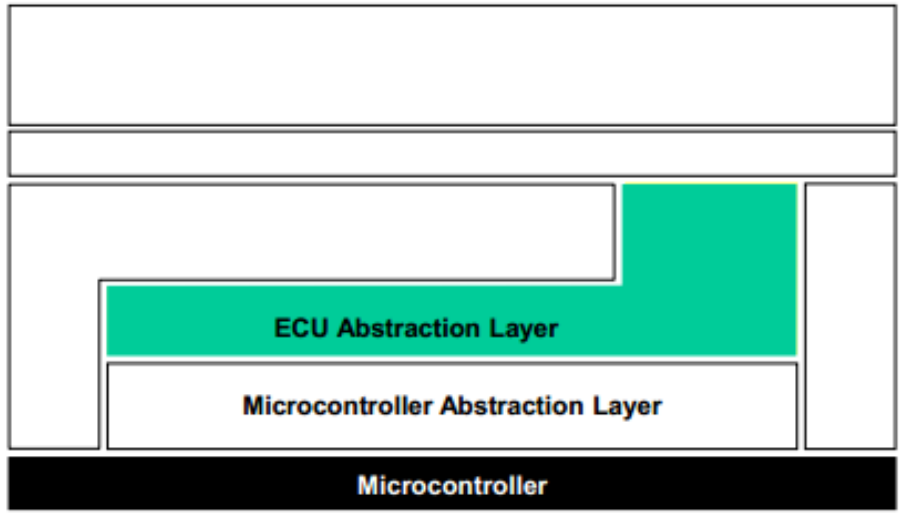
Upper Interface: standardized and µC independent



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

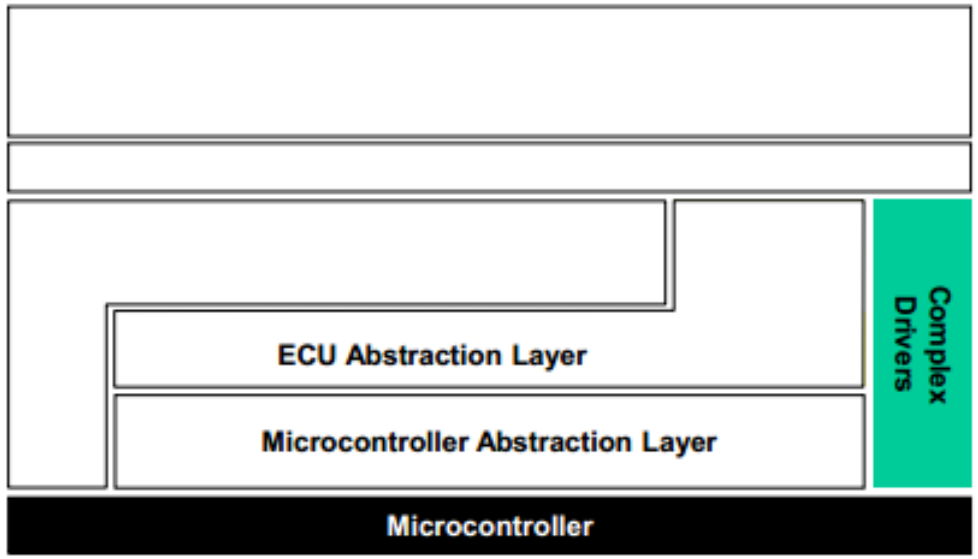
Properties

Implementation: µC independent, ECU hardware dependent

Upper Interface: µC and ECU hardware independent

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 (Wdg manager)
- Cryptographic Services (Crypto Service Manager)

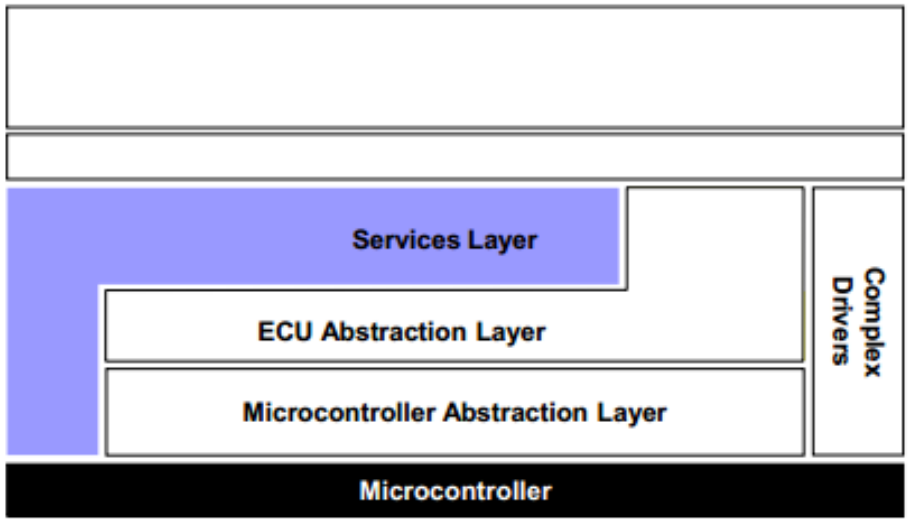
Task

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.

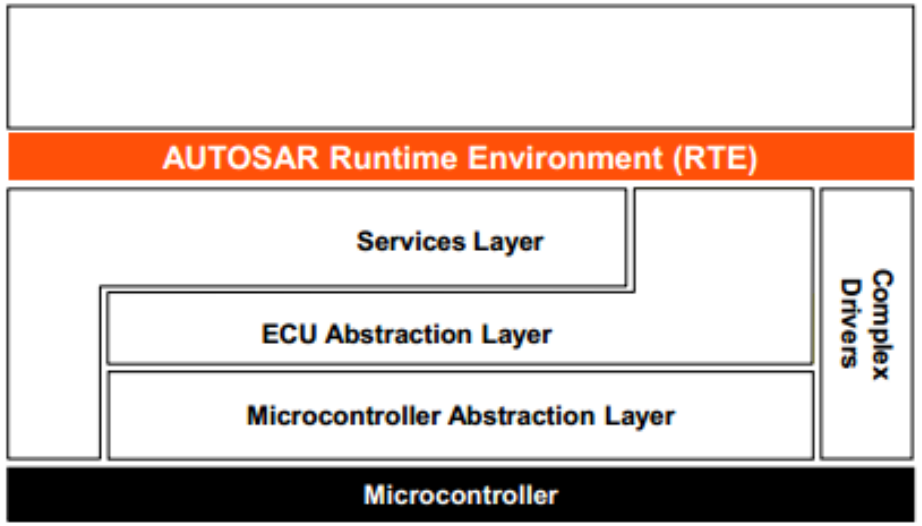
Task

Make AUTOSAR Software Components independent from the mapping to a specific ECU.

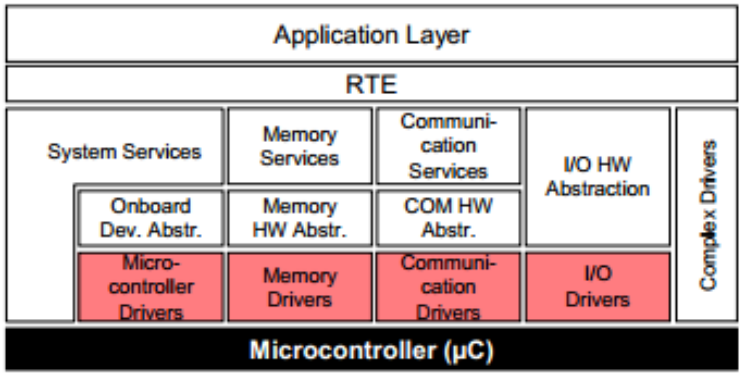
Properties

Implementation: ECU and application specific (generated individually for each ECU)

Upper Interface: completely ECU independent

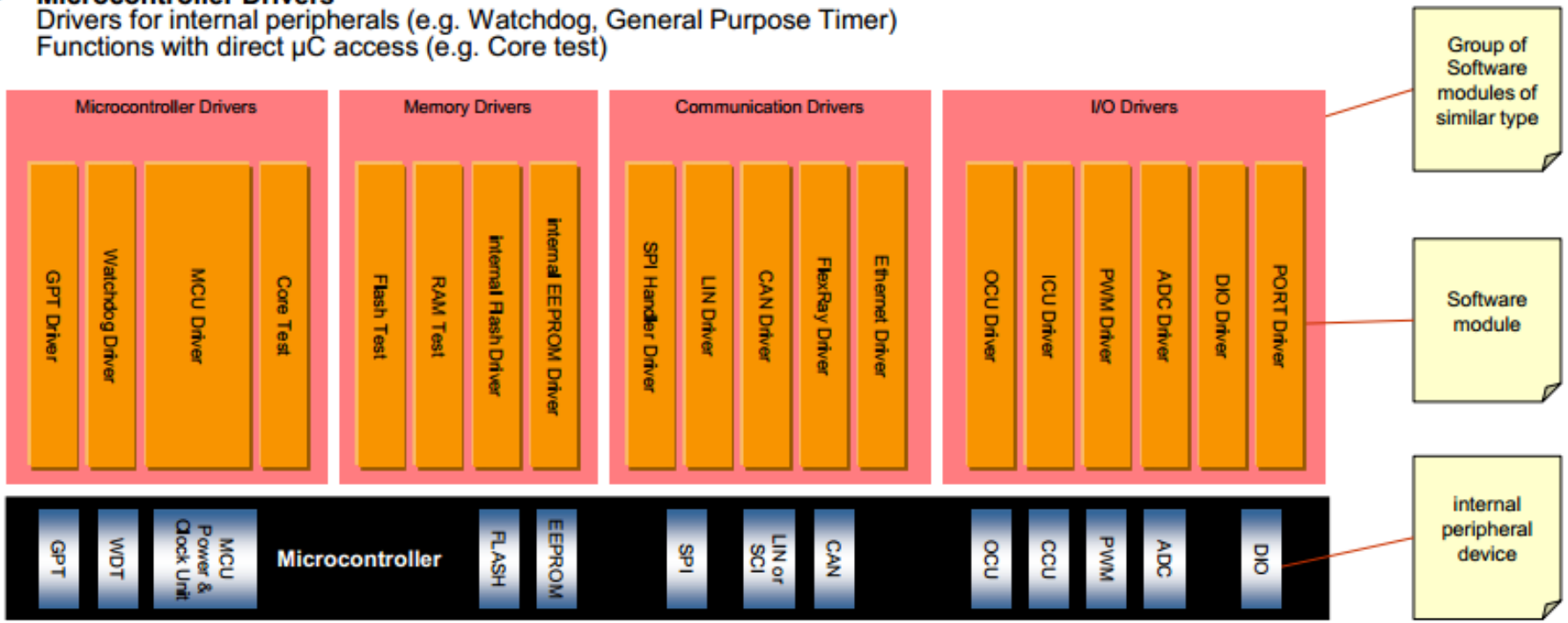


Architecture – Content of Software Layers
Microcontroller Abstraction Layer

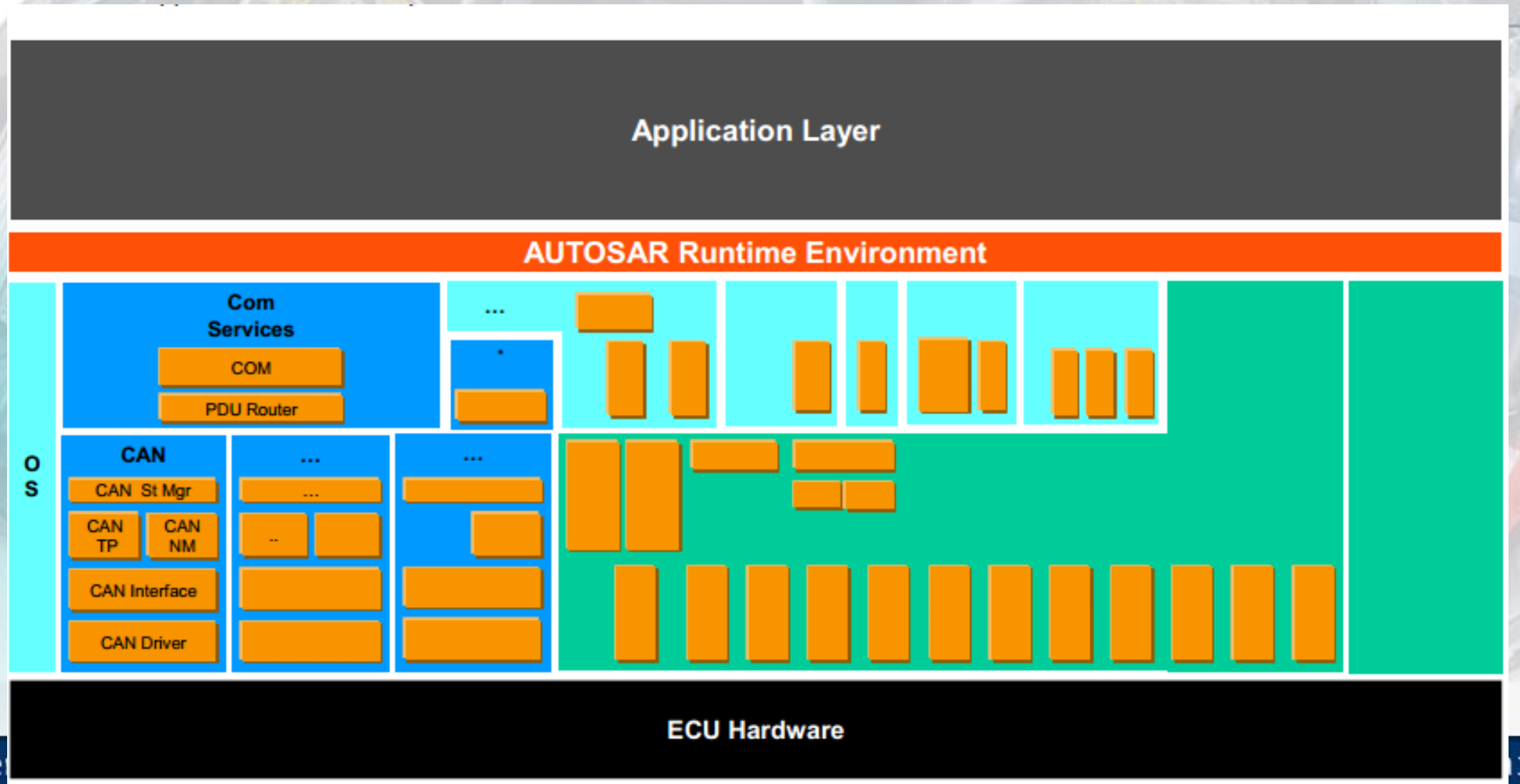


The **µC Abstraction Layer** consists of the following module groups:

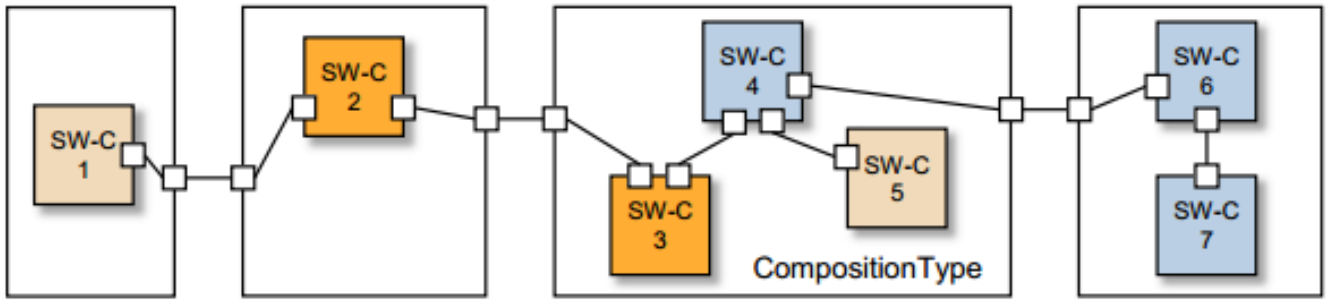
- **Communication Drivers**
  Drivers for ECU onboard (e.g. SPI) and vehicle communication (e.g. CAN).
  OSI-Layer: Part of Data Link Layer
- **I/O Drivers**
  Drivers for analog and digital I/O (e.g. ADC, PWM, DIO)
- **Memory Drivers**
  Drivers for on-chip memory devices (e.g. internal Flash, internal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

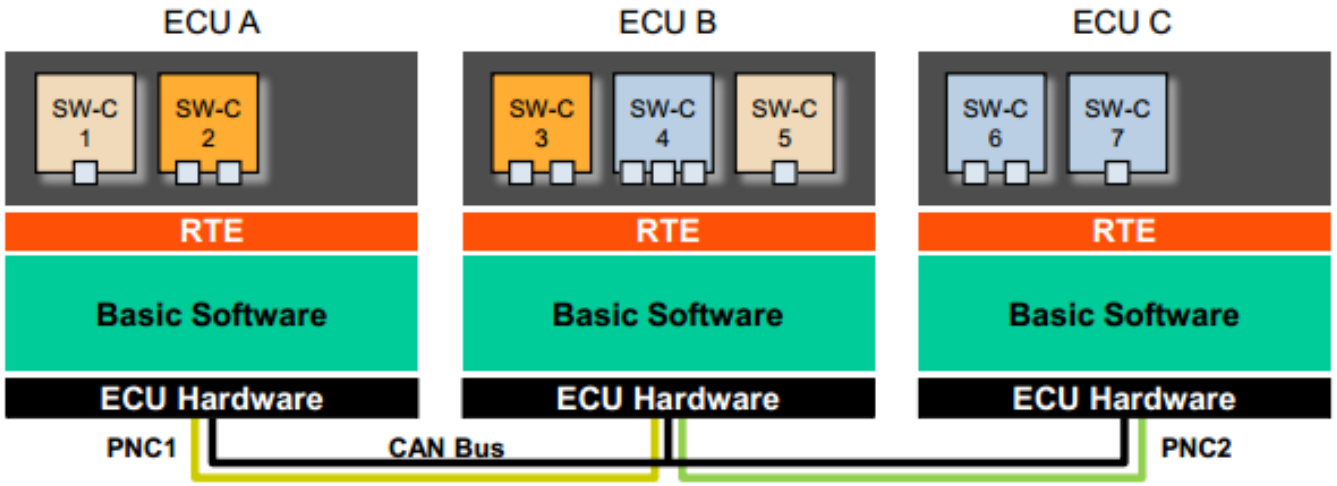


CAN

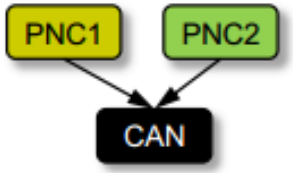

Application Layer
AUTOSAR Runtime Environment
Com Services
COM
PDU Router
...
CAN
CAN St Mgr
CAN TP
CAN NM
CAN Interface
CAN Driver
...
...
OS
ECU Hardware

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





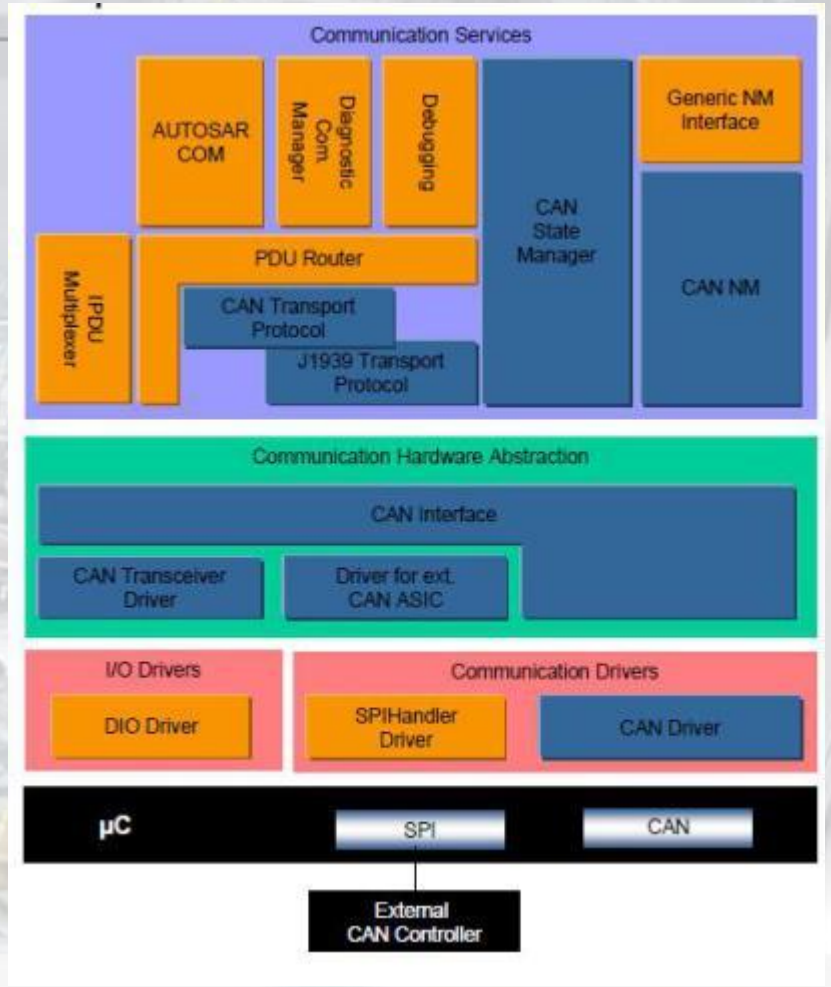
- Here both Partial Networks map to one CAN bus.
- One Partial Network can also span more than one bus.



CAN Communication:

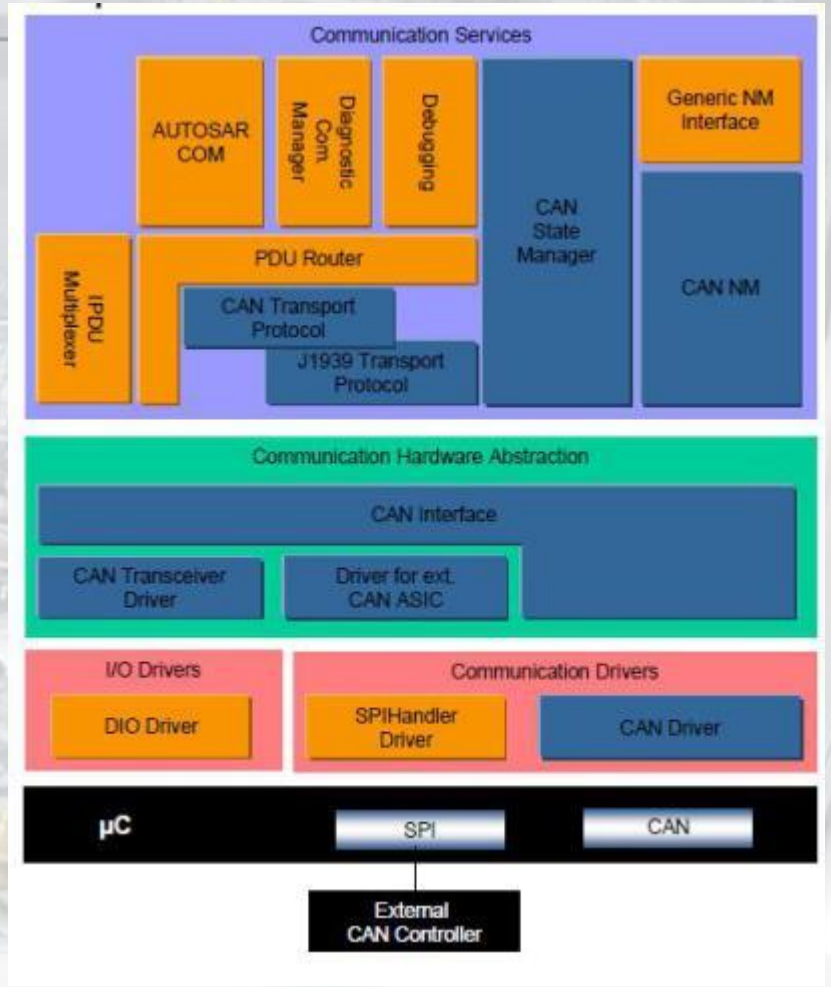
- **Application Layer and RTE**

Applications written in the context of AUTOSAR consist of components. These components communicate with each other via ports (component view). The communication between two components can consist of a single (AUTOSAR) signal or a whole signal group. From the view of the AUTOSAR SWC it is not known at implementation time, which communication media is used. All bus specific replications of send requests by a SWC to underlying layers and bus specific timing behavior must be done by COM or by the appropriate bus interfaces and drivers. The RTE uses the capability to send and receive signals of AUTOSAR COM. VFB's send modes corresponding to the transfer property of a signal and the transmission mode of an I-PDU.



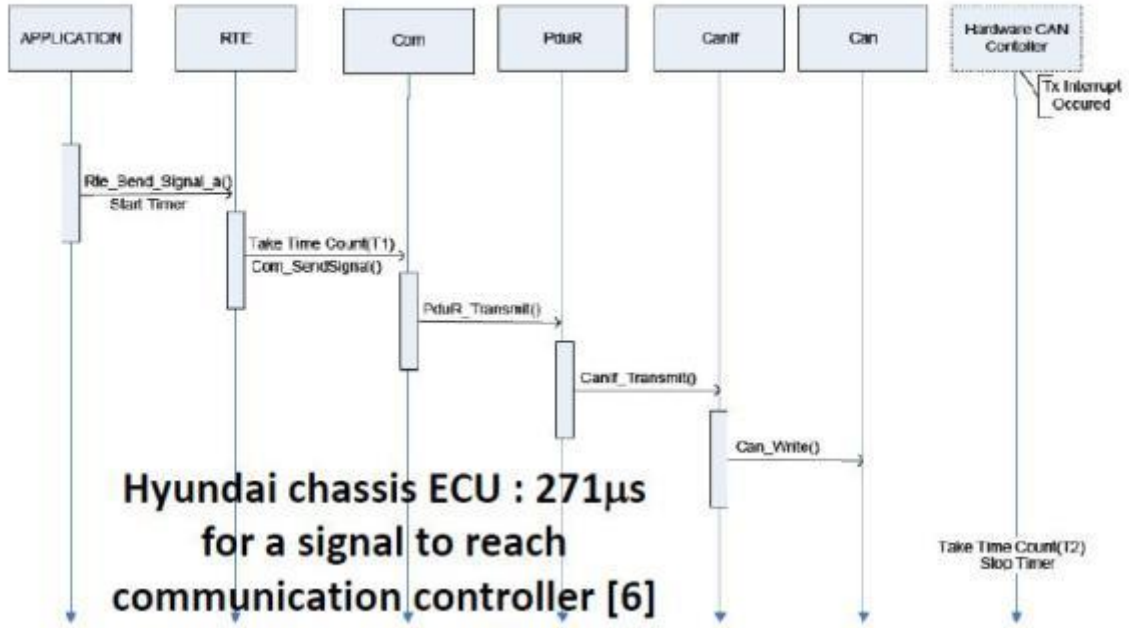
Transmission Modes and Transmission Model Selection

- COM shall provide different transmission modes for each I-PDU.
- **Periodic:** transmissions occur indefinitely with a fixed period between them
- **Direct / n-times:** event driven transmission with n-1 repetitions
- **Mixed:** periodic transmission with direct/n-times transmissions in between
- **None:** no transmission



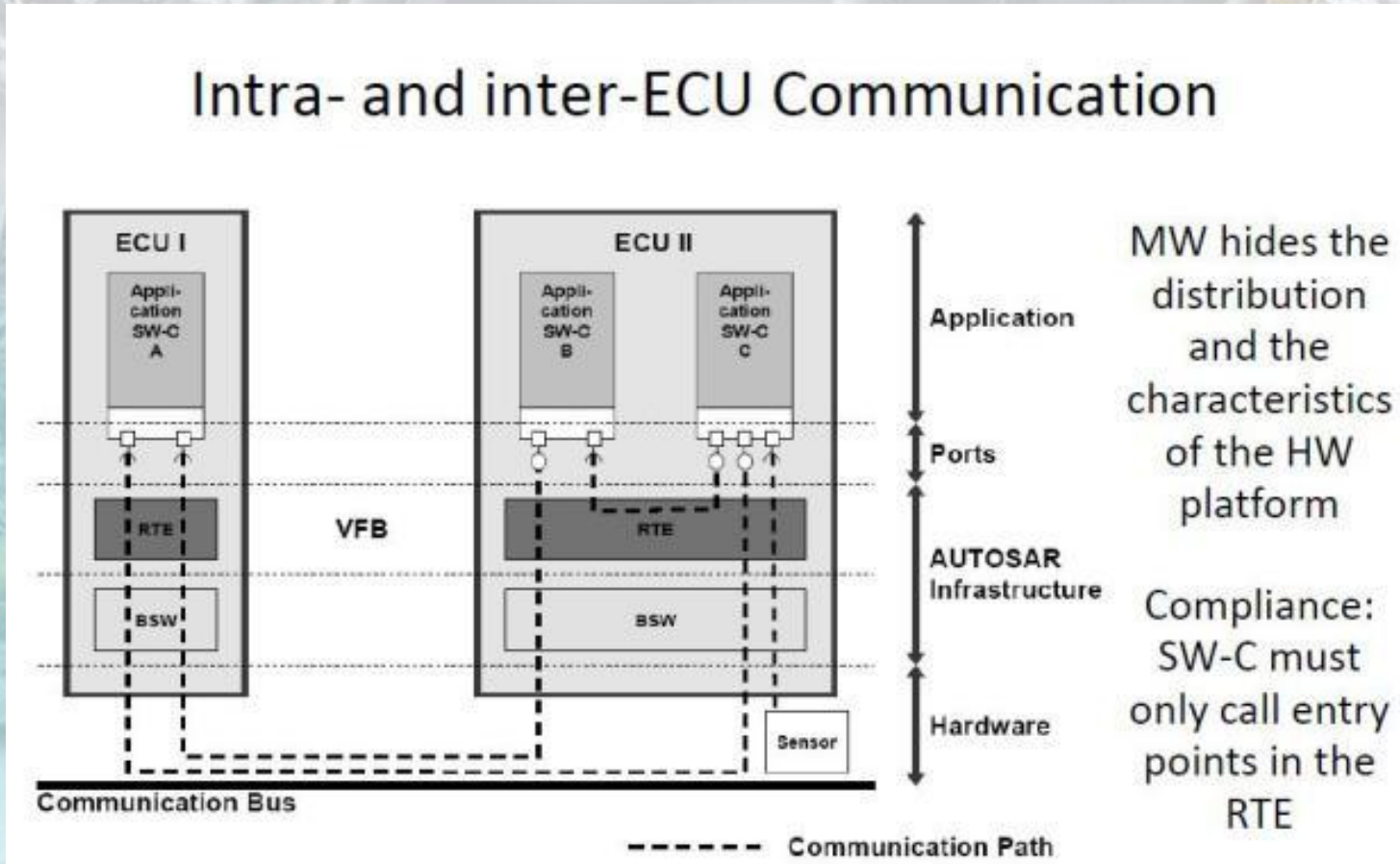
Transmission Modes and Transmission Model Selection (Cont.)

Sending a signal through the CAN communication stack [6]



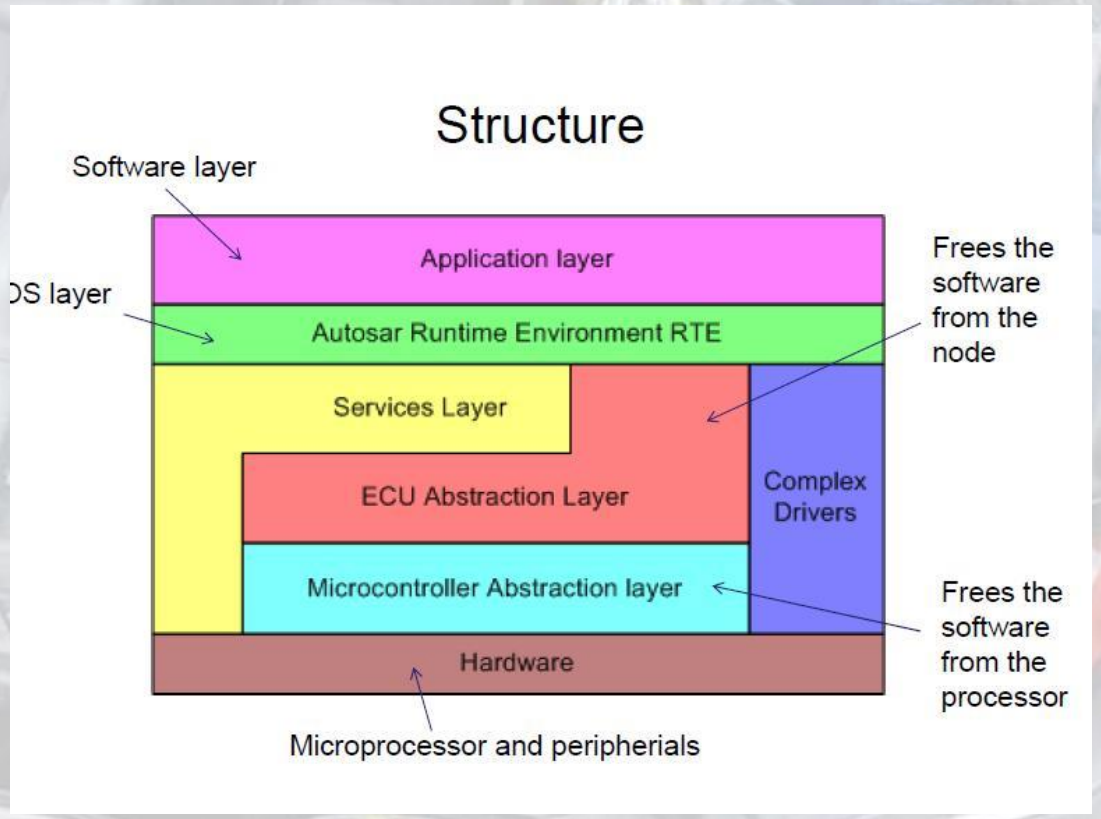
Hyundai chassis ECU : 271µs for a signal to reach communication controller [6]

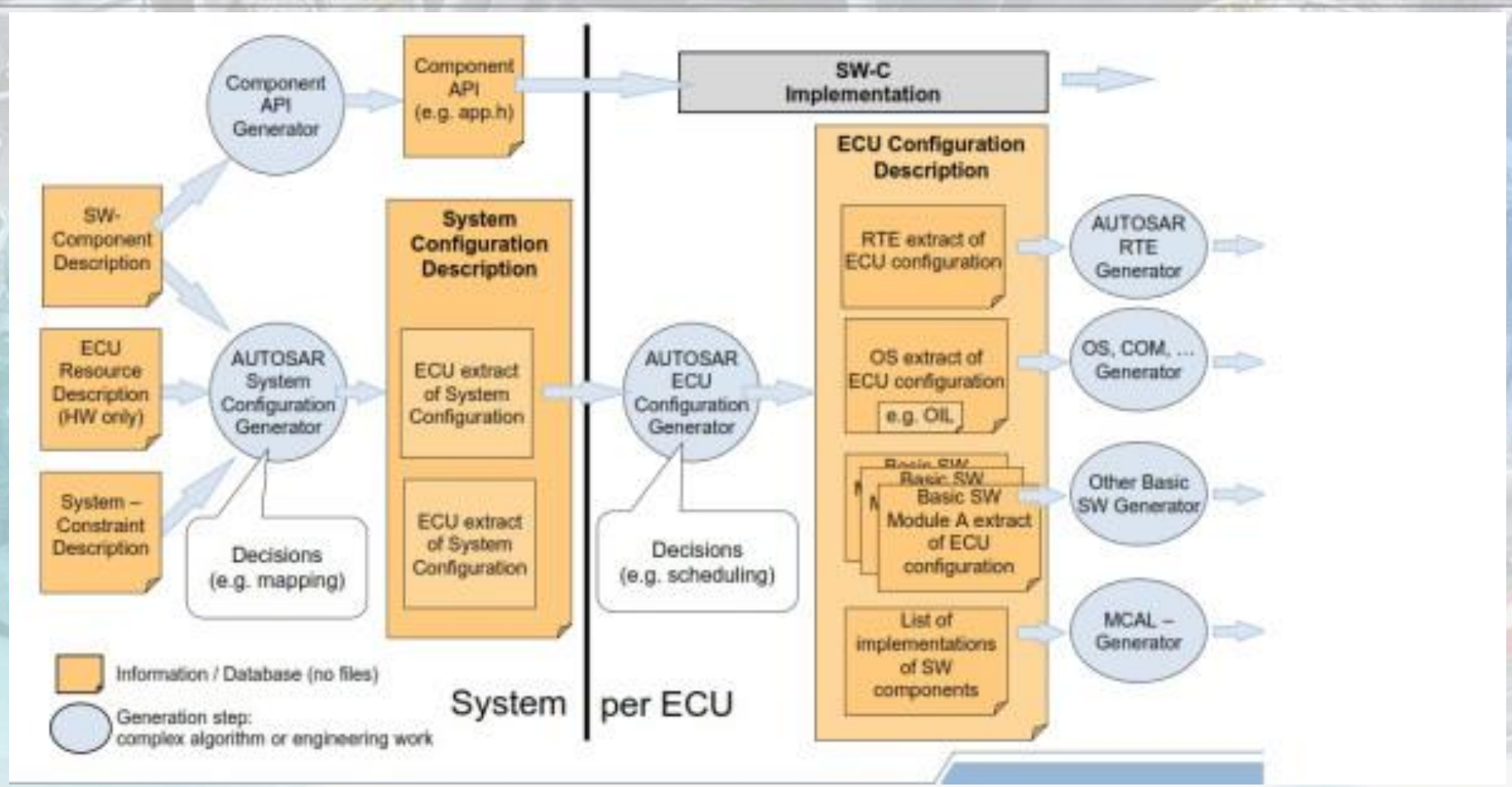
In Autosar, two types of SWC communication **Intra & Inter,**



Autosar Interfaces:

- An AUTOSAR Interface describes the data and services required or provided by a component and is specified and implemented according to the AUTOSAR Interface Definition Language. An AUTOSAR Interface is partly standardized within AUTOSAR, e.g. it may include OEM specific aspects. The use of AUTOSAR Interfaces allows software components to be distributed among several ECUs. The RTEs on the ECUs will take care of making the distribution transparent to the software components.




Component API Generator
Component API (e.g. app.h)
SW-C Implementation
SW-Component Description
ECU Resource Description (HW only)
System – Constraint Description
AUTOSAR System Configuration Generator
Decisions (e.g. mapping)
System Configuration Description
ECU extract of System Configuration
ECU extract of System Configuration
AUTOSAR ECU Configuration Generator
Decisions (e.g. scheduling)
ECU Configuration Description
RTE extract of ECU configuration
OS extract of ECU configuration
e.g. OIL
Basic SW Module A extract of ECU configuration
List of implementations of SW components
AUTOSAR RTE Generator
OS, COM, … Generator
Other Basic SW Generator
MCAL – Generator
Information / Database (no files)
Generation step: complex algorithm or engineering work
System
per ECU

Client-Server Communication:

- A widely used communication pattern in distributed systems is the client-server pattern, in which the server is a provider of a service and the client is a user of a service.
  The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server waits for incoming communication requests from a client, performs the requested service, and dispatches a response to the client's request. The direction of initiation is used to categorize whether an AUTOSAR Software Component is a client or a server. A single component can be both a client and a server, depending on the software realization. The client can be blocked (synchronous communication) or non-blocked (asynchronous communication), respectively, after the service request is initiated until the response of the server is received. The image gives an example how client-server communication for a composition of three software components and two connections is modeled in the VFB view.

Sender-Receiver Communication:

- The sender-receiver pattern gives solution to the asynchronous distribution of information, where a sender distributes information to one or several receivers. The sender is not blocked (asynchronous communication) and neither expects nor gets a response from the receivers (data or control flow), i.e. the sender just provides the information and the receivers decides autonomously when and how to use this information. It is the responsibility of the communication infrastructure to distribute the information.
  The sender component does not know the identity or the number of receivers to support transferability and exchange of AUTOSAR Software Components. The image illustrates an example how sender-receiver communication is modeled in the AUTOSAR VFB view.
- The central structural element in AUTOSAR is the COMPONENT. A component has well-defined ports, through which it interacts with other components. A port always belongs to exactly one component. The AUTOSAR Interface concept defines the services or data that are provided on or required by a port of a component. The AUTOSAR Interface can either be a Client-Server Interface (defining a set of operations that can be invoked) or a Sender-Receiver Interface, which allows the usage of data-oriented communication mechanisms over the VFB.
  A port is either a PPort or an RPort. A PPort provides an AUTOSAR Interface while an RPort requires one.
- When a PPort of a component provides an interface, the component to which the port belongs provides an implementation of the operations defined in the Client-Server Interface respectively generates the data described in a data-oriented Sender-Receiver Interface.
  When an RPort of a component requires an AUTOSAR Interface, the component can either invoke the operations when the interface is a Client-Server Interface or alternatively read the data elements described in the Sender-Receiver Interface.

CAN Driver

- **CAN Driver:** The CAN Driver is part of the lower layer and offers the CAN Interface uniform interfaces to use. It hides hardware specific properties of the CAN Controller as far as possible. The CAN Driver performs the hardware access and provides a hardware independent API to the upper layer, the CAN interface (CanIf). Services for initiating transmission are offered by the CAN Driver and it calls the callback funtions of the CanIf module for notifying events hardware independently. In addition there are services provided by the CAN Driver module to control the state of all CAN controller belonging to the same CAN hardware unit. A CAN controller serves exactly one physical channel. A detailed description of the CAN bus is given in [30]. A CAN hardware unit is represented by one CAN Driver and either on chip or an external device. It may consist of one or multiple CAN controllers of the same type and one or multiple CAN RAM areas [29]. A single CAN Driver module can handle multiple CAN controllers if they belong to the same hardware unit. If an L-PDU shall be transmitted, the CAN Driver writes the L-PDU in a buffer inside the CAN controller hardware and if an L-PDU is received, the CAN Driver module calls the RX indication callback funtion with the L-PDUs ID, the DLC (see: ch. 2.3) and with a pointer to the L-SDU. The CAN Driver can access hardware resources and converts the given information for transmission into a hardware speci c format and triggers the transmission. The CAN Driver module offers the CanIf services to control the state of the CAN. Controllers by callback functions for bus-off and wake-up events. It implements the interrupt service routines for all CAN hardware unit interrupts that are needed. While startup the CAN Driver initializes static variables including flags, sets common settings for the complete CAN hardware unit and sets CAN controller specific settings for each CAN controller.

Communication Manager (ComM):

- **Communication Manager (ComM):** The ComM is a resource manager which encapsulates the control of the underlying communication services. It controls the basic software modules related to communication and coordinates the bus communication access requests. The ComM shall simplify the usage of the bus communication stack for the user. It shall offer an API for disabling the sending of signals, shall be able to control more than one communication bus channel of an ECU and shall simplify the resource management by allocating all resources necessary for the requested communication mode. The COM Manager (ComM) controls the starting and stopping of sending and receiving I-PDUs via AUTOSAR COM. The NM is used by the ComM to synchronize the control of communication capabilities across the network.

CAN/FlexRay/LIN Bus State Manager:

- **CAN/FlexRay/LIN Bus State Manager:** The actual bus states are controlled by the corresponding Bus State Manager. The actural states of the bus corresponds to a communication mode of the ComM. The ComM requests a specific communication mode from the state manager and the state manager shall map the communication mode to a bus state.
- E.g. the comM uses the API of the CanSM to request communication modes of CAN neworks. The CanSM uses the API of COM to controll CAN related PDU groups and it communicates with the CanIf to conrol the operating modes of the CAN controllers and to get noti ed by the CanIf about peripheral events.

Network Management Modules (NM)

- **Network Management Modules (NM)** The Generic Nework Management Interface adapts the ComM to the bus specific network management modules. It provides an inteface to the ComM and uses services of the network management modules. The bus specific network management modules are CAN NM, FlexRay NM and LIN NM. The AUTOSAR NM Interface can only be applied to communication systems that support broadcast communication and bus-sleep mode. For network management data exchange the PDU Router module is bypassed.

DCM(Diagnostic Communication Manager):

- **DCM(Diagnostic Communication Manager**: The main purpose of the DCM is providing a common API for diagnostic services. It is used while development, manufactoring or service by external diagnostic tools [25]. In gure 3.5 there is an overview of the communication over the DCM. The DCM performs the scheduling of diagnostic PDUs. It acts as a user by requesting full communication from the ComM if diagnostic shall be performed.

DCM(Diagnostic Communication Manager): (Cont.)

- A more likely scenario is the injection of isolated AUTOSAR-"islands" in a project. This scenario gives few of AUTOSAR´s intended benefits in the short perspective but gives valuable AUTOSAR experience.

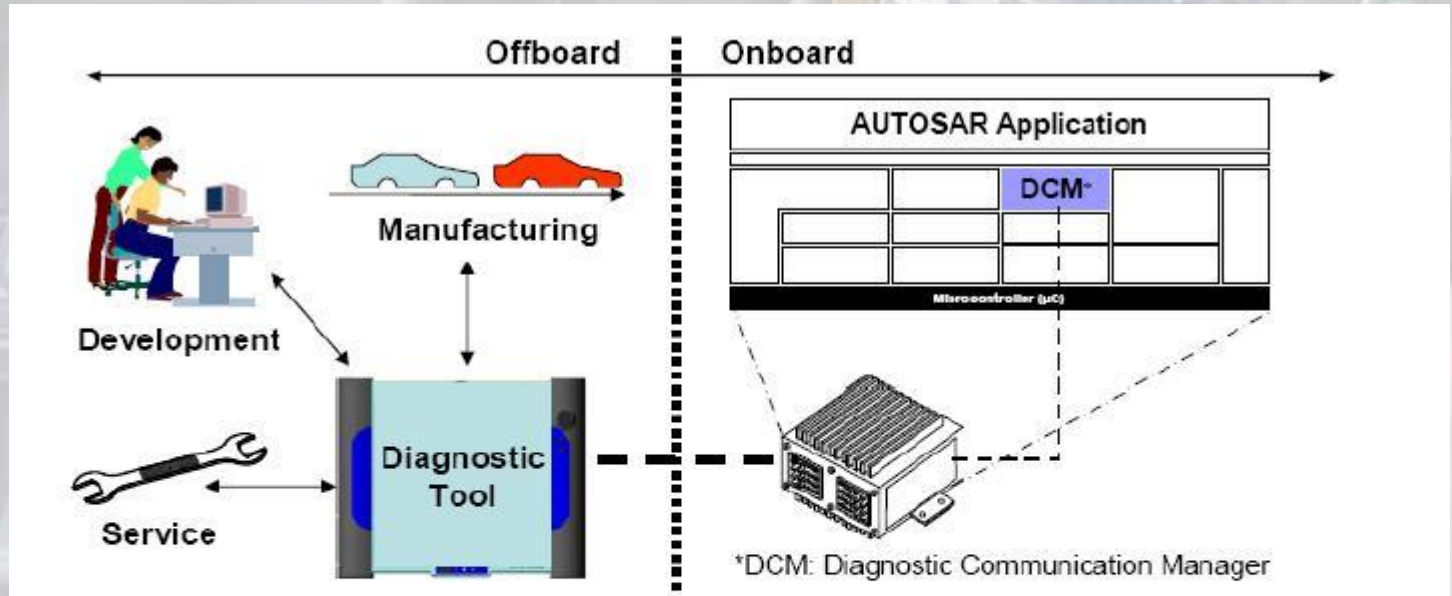


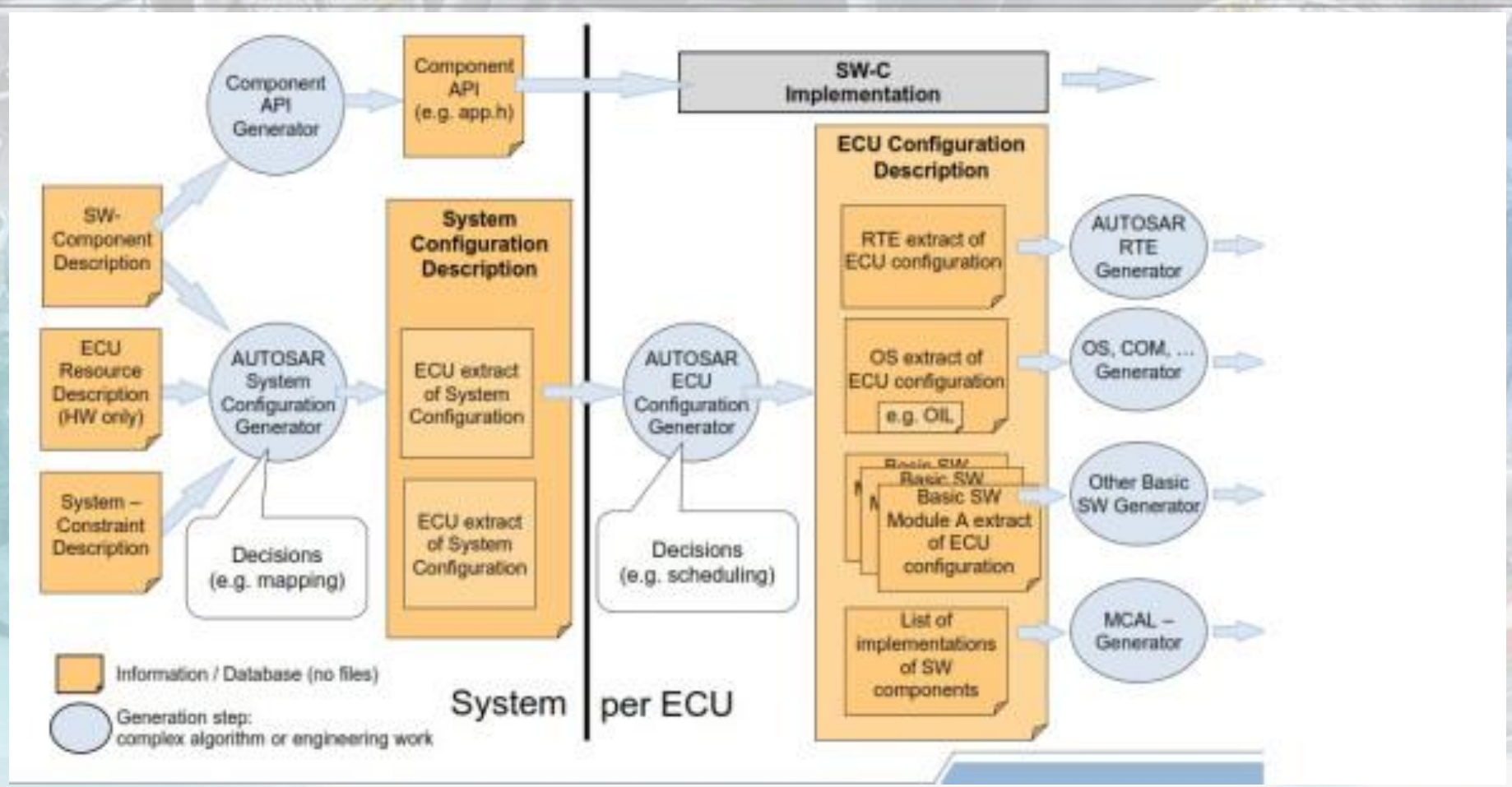
Fig. 3.5. Overview of the communication between the external diagnostic tools and the onboard AUTOSAR Application [25]

How they have standardised the Autosar Basic Software modules(BSW)?

- In Every module, there are standardised functionality for example ADC, The main function of ADC is to convert Analog to Digital conversion. So, we can develop standard functions but the channel, Bit resolution, Interrupts might be changing based upon the microcontroller and hardware pins. In Each module, we can divide into two types configuration files for hardware related information(like Channels, Groups, Pins, Pin Direction, Resolution) and Standard functions for modules main functionality. In each BSW (Basic Software module) Configuration files which are configured using tools(Eg: EBTresos, EcuSpectrum, DaVinci Configurator) we can generated configuration files and Standardized function which are defined as per autosar SWS specification.

How to generate the System Description files and ECU Description files?

- Load the Dbc(Can Database) or LDF(Lin Description File) or Fibex (Field Bus Exchange) in Configuration tool and Configure the missing parameter as per System Description Template and extract the System Description arxml file using Export option. We can generate the ECU Description file using ECU extract option after loading the System Description file. After the extraction of ECU Description arxml load it configuration tool and configure the BSW modules as per autosar SWS Specification document.
  - **System Configuration Description:**
    includes all system information and the information that must be agreed between different ECUs
  - **System Configuration Extractor:**
    extracts the information from the System Configuration Description needed for a specific ECU
  - **ECU extract:**
    is the information from the System Configuration Description needed for a specific ECU
  - **ECU Configuration Description:**
    all information that is local to a specific ECU the runnable software can be built from this information and the code of the software component


Component API Generator
Component API (e.g. app.h)
SW-C Implementation
SW-Component Description
ECU Resource Description (HW only)
System – Constraint Description
AUTOSAR System Configuration Generator
Decisions (e.g. mapping)
System Configuration Description
ECU extract of System Configuration
ECU extract of System Configuration
AUTOSAR ECU Configuration Generator
Decisions (e.g. scheduling)
ECU Configuration Description
RTE extract of ECU configuration
OS extract of ECU configuration
e.g. OIL
Basic SW Module A extract of ECU configuration
List of implementations of SW components
AUTOSAR RTE Generator
OS, COM, … Generator
Other Basic SW Generator
MCAL – Generator
Information / Database (no files)
Generation step: complex algorithm or engineering work
System
per ECU

AUTOSAR TOOLS

Implementer	BSW Implementation	BSW Configurator	RTE Generator	System Tooling
ArcCore	Arctic Core –	BSW Builder	RTE Builder	SWC Builder & Extract Builder
	CUBAS, iSolar [6]	CUBAS, iSolar [7]	CUBAS, iSolar [7]	Unknown
Continental	Yes	Yes	Yes	Yes
dSPACE	No	No	SystemDesk RTE Generator	SystemDesk
Elektrobit	EB tresos AutoCore	EB tresos Studio	EB tresos Studio	No
ETAS	Yes	Yes	RTA	ISOLAR-A
Freescale	Yes [8]	No	Yes [8]	Unknown
Dassault Systèmes	No	GCE	RTEG	AAT
KPIT Cummins	Yes	ECU Spectrum Toolchain	ECU Spectrum Toolchain	ECU Spectrum Toolchain
Mecel	Yes	Yes	Yes	Unknown
Mentor Graphics	Volcano VSTAR	Volcano VSTAR	Volcano VSTAR	Volcano Systems Architect
OpenSynergy	COQOS (OS & SchM)	COQOS	COQOS	No
Renesas Electronics	Yes	No	No	No
see4sys	Yes	Yes	Yes	ECU-Designer
Vector Informatik GmbH	MICROSAR	DaVinci Configurator Pro	MICROSAR Rte Generator	DaVinci System Architect

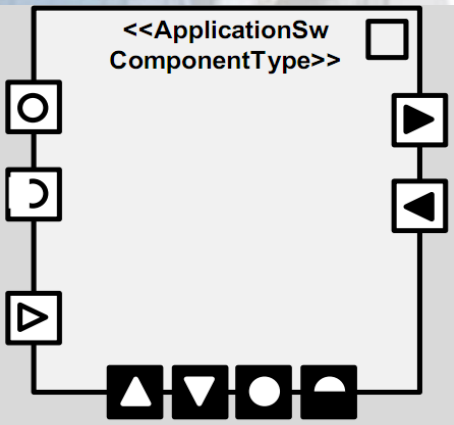
AUTOSAR Software Component

SWC Types

- Application software component
- Sensoractuator software component
- Parameter software component
- Composition software component
- Service proxy software component
- Service software component
- Ecuabstraction software component
- Complex driver software component
- Nvblock software component

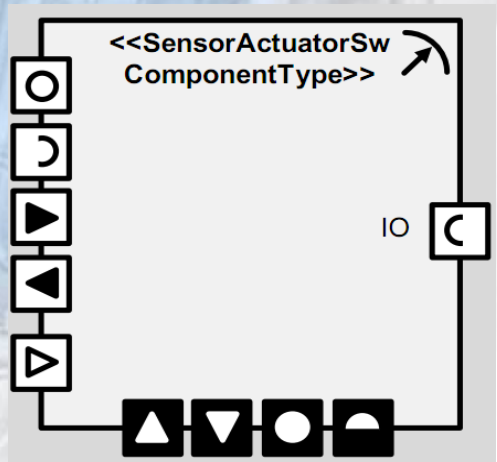
Application SWC

- is an atomic software component that carries out an application or part of it. It can use ALL AUTOSAR communication mechanisms and services
- Application SWCs are our main building blocks



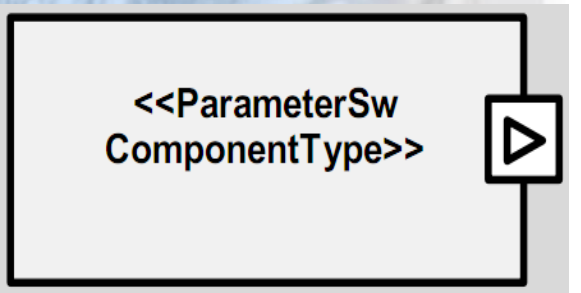
Sensoractuator software component

- is an atomic SWC that handles the specifics of sensors or actuators. It directly interacts with the ecu-abstraction



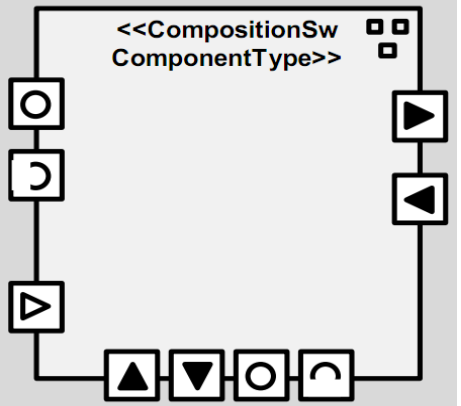
Parameter Software Component

- Atomic SWC
- it provides parameter values. They can be fixed data, const or variable. It allows access to fixed data or calibration data
- They don't have an internal behavior
- They only have PPorts of ParameterInterfce type
- Need to be on the same ecu as the SWCs accessing them since a parameter SWC represents the memory containing the calibration parameter



Composition software components

- a composition of atomic or non-atomic (composite) software components which is an encapsulation
- Non-atomic
- Has no binary footprint
- Primarily used to abstract a bunch of SWCs from other SWCs on VFB level

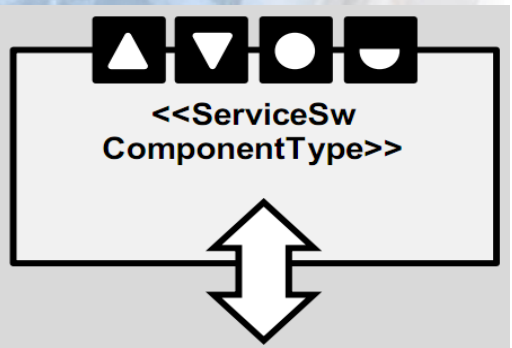


Service proxy SWC

- responsible for distribution of modes through the system (inter-ECU) since AUTOSAR's mode switch system only supports intra-ECU communication
- each ECU will need a copy of this since service proxy SWCs are to take care of informing ECUs of the mode changes

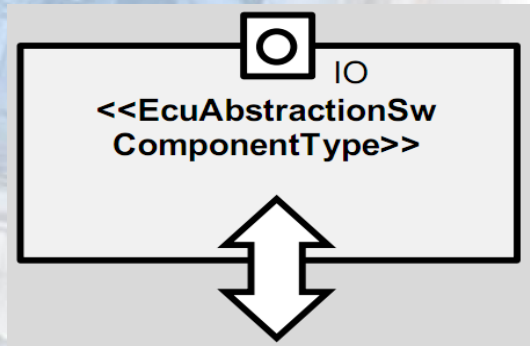
Service software component

- provides services specified by AUTOSAR through interfaces specified by AUTOSAR. This component may interact directly with modules from BSW
- Represents the different BSW Module services in the VFB view



ECU-abstraction software component

- it provides access to the ECU's IOs. It can interact with certain BSW modules. (again that arrow thingy at the bottom). The services are usually provided through PPorts and are used by SensorActuator SWCs
- BSW layer
- Represents the ECUAbstraction layer and its services
- The only SWC that is allowed to access IO ports



SWC Connectors

- Assembly connector: all the connections between SWCs are of this type
- Delegation connector: in a composition SWC, the ports of the inner SWCs that need to be visible from the outside of the composition, i.e. the composition SWC's ports need to be connected to delegation connectors

Example of access pattern to sensors and actuators:

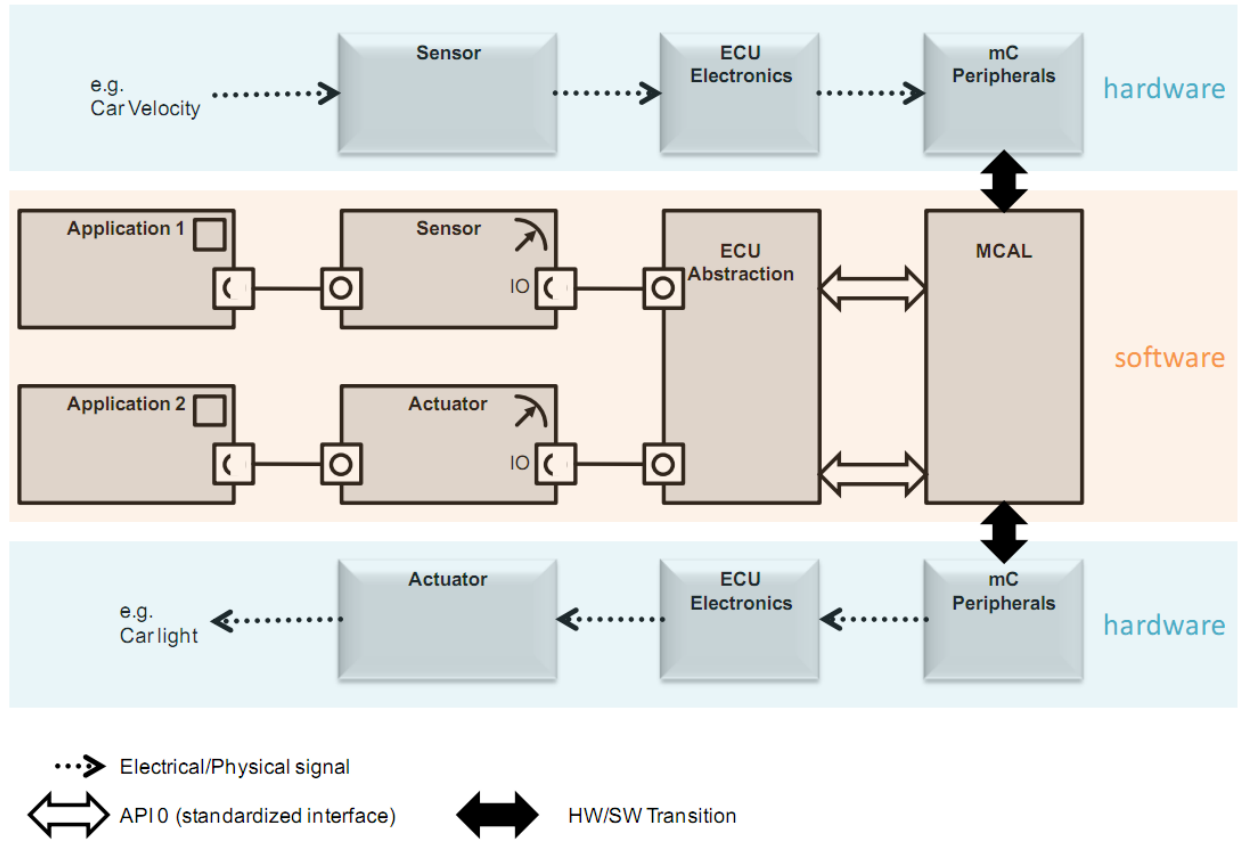
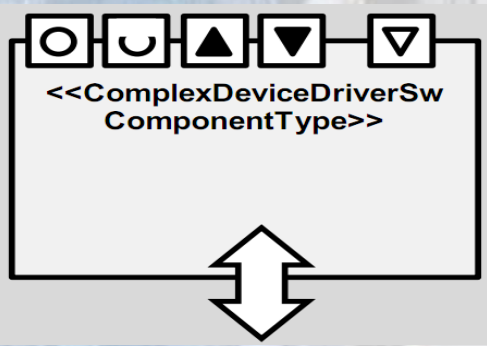


Figure 6.1: Signal conversions between physical signals and software signals

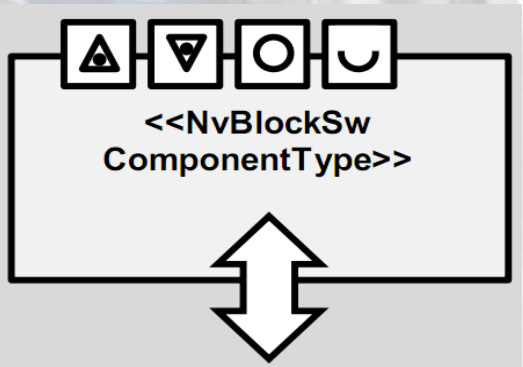
Complex driver software component

- it generalizes the ECUAbstraction component. It can define ports to interact with components in specific ways and can also directly interact with BSW modules
- Complex device drivers can use BSW services
- Complex device drivers exist to fulfill certain needs:
  - Implementing a complex application that cannot be otherwise implemented due to the AUTOSAR BSW layered architecture
  - Timing critical applications
  - Non-AUTOSAR applications within AUTOSAR ECU



NVBlock SWC

- it allows SWCs to access NV data
- It represents the Nvmanager from BSW layer

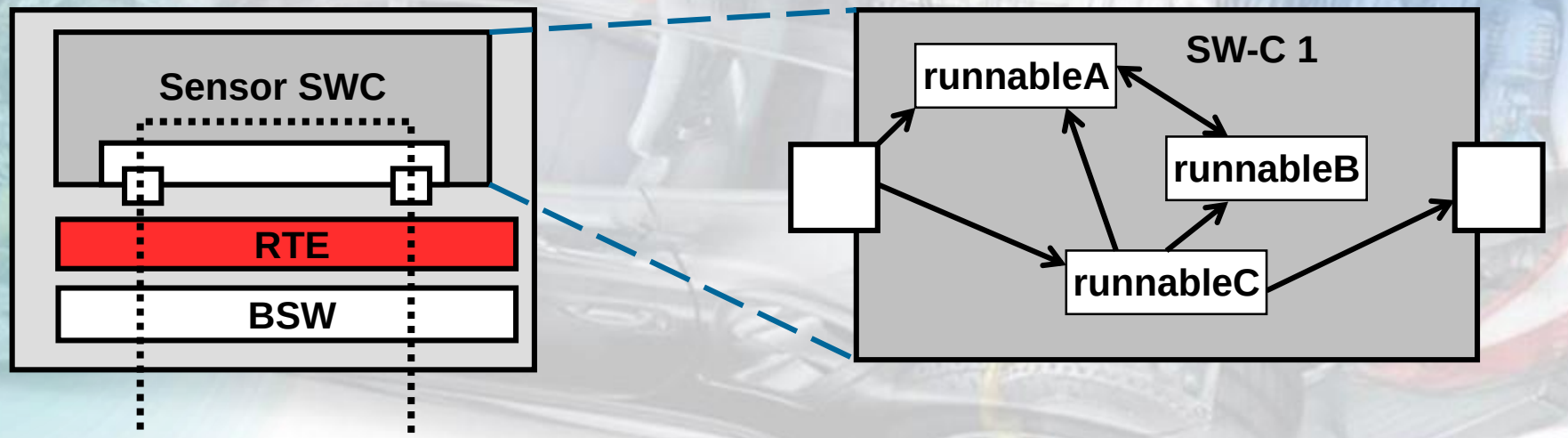


SWC elements

- Ports
  - PPort → provide port
  - Rport → require port
  - PRPort → provide require port
- Internal Behavior
  - Runnables
    - RTE Events
    - InterrunnableVaribales
- Implementation (source or object code)

SW Components and Runnables

- SW-Components
  - atomic components with respect to *mapping*
  - provided by one supplier
- Runnables
  - atomic components with respect to *execution*
  - attached to different OS Tasks

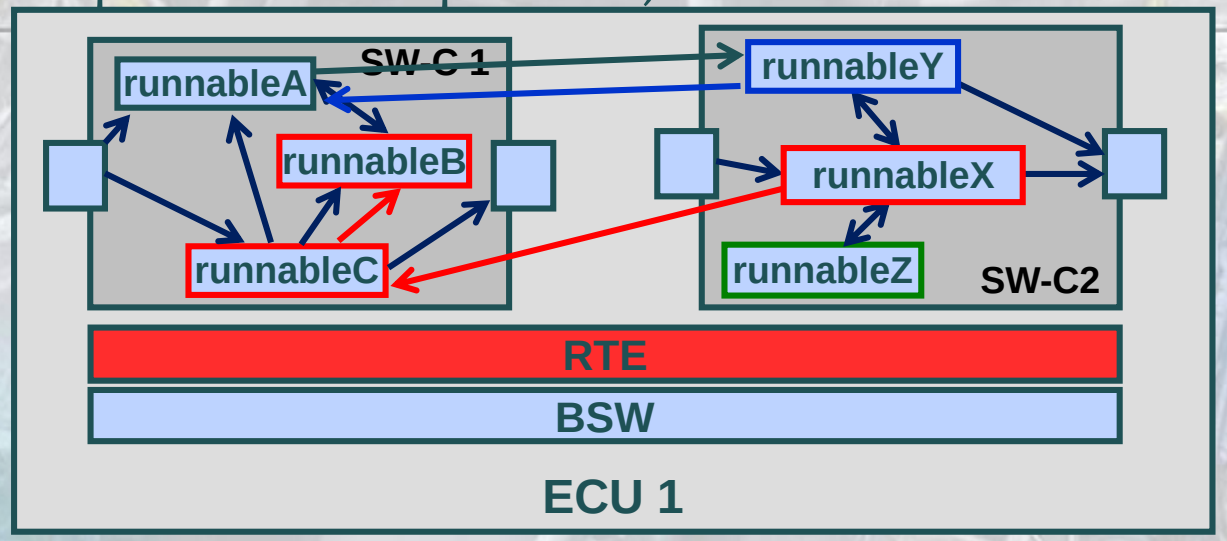


SWC Description and Elements

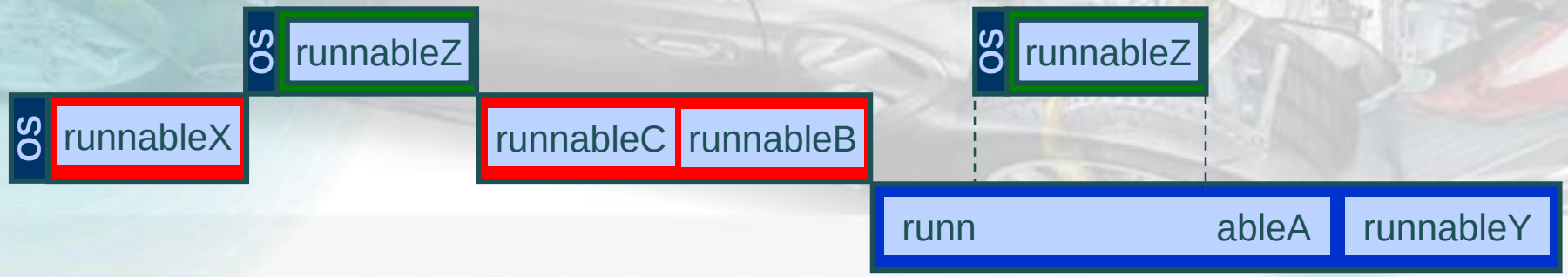

```
component application appswc{
    ports{
        sender swcpport provides sri1
        receiver swcrport requires sri2

    }
}
internalBehavior appswcib for appswc{

    runnable appswcrun1 [0.0]{
        symbol "appswcrun1"

        dataReceivedEvent swcrport.data2 as appswcgotit

    }

}
implementation appswcimpl for appswc.appswcib{
    language c
    codeDescriptor "src"
}
```

Runnables and Tasks

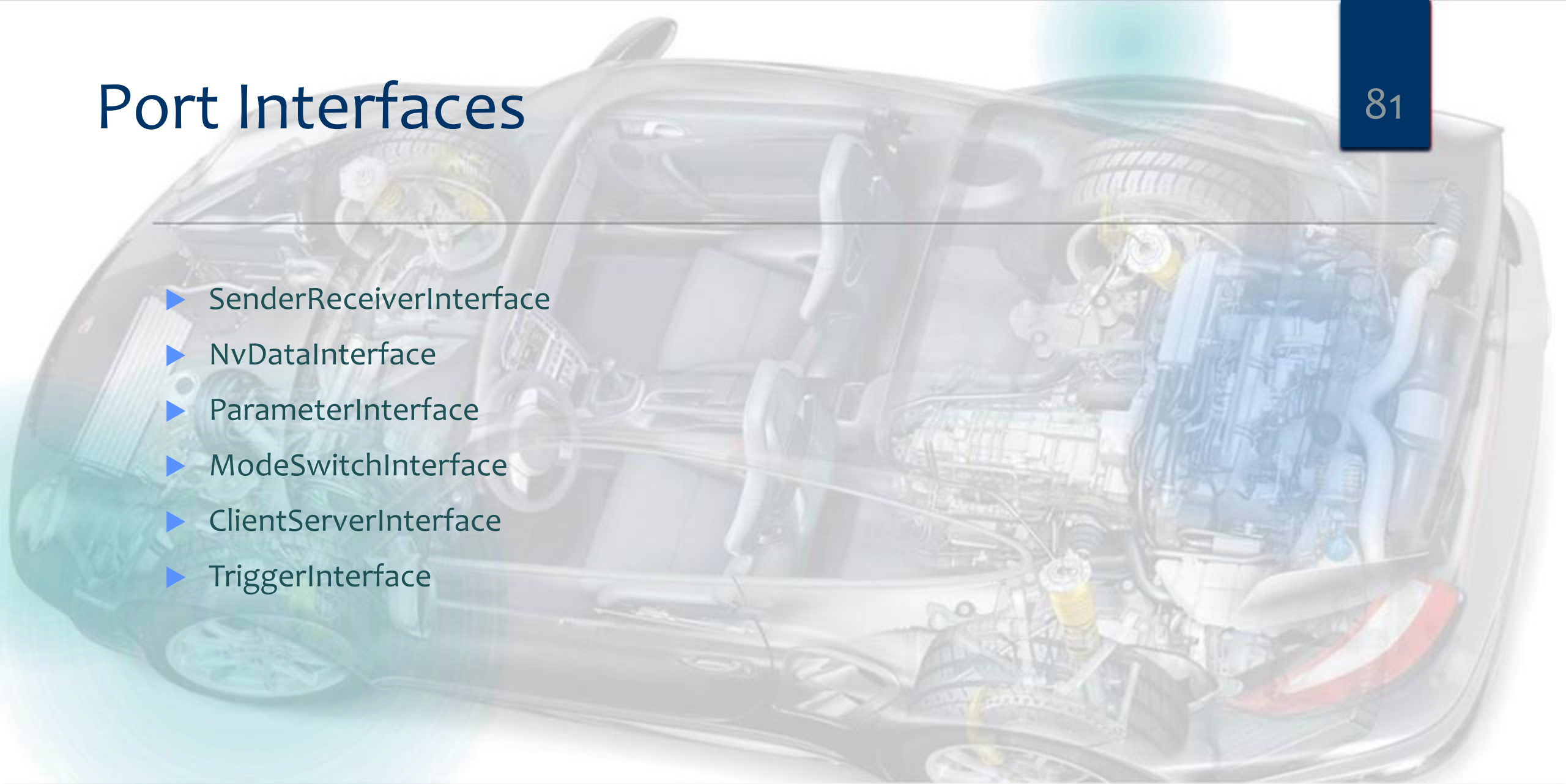
SW architecture example: 2 SW components, 6 runnables



Schedule and timing dependencies



Port Interfaces



- SenderReceiverInterface
- NvDataInterface
- ParameterInterface
- ModeSwitchInterface
- ClientServerInterface
- TriggerInterface

Supported value encoding

Supported value encodings that can be used inside an AUTOSAR port:

- 2C: two's complement
- IEEE754: floating point numbers
- ISO-8859-1: ASCII-Strings
- ISO-8859-2: ASCII-Strings
- WINDOWS-1252: ASCII-String
- UTF-8: UCS Transformation format 8
- UCS-2: universal character set 2
- NONE: unsigned integer
- BOOLEAN: this represents an integer to be interpreted as boolean

The following types are applicable if the port typed by the interface is not a service port and hence, is a data port

SenderReceiverInterface

- Allows for the specification of the typically asynchronous communication pattern where a sender provides data that is required by one or more receivers (1:n or n:1)
- For SenderReceiverInterface, n:m while n or m are bigger than one is not possible
- Can invalidate receiving data
- handleInvalidEnum
  - dontInvalidate: invalidation is switched off
  - Keep: the error code returned by the RTE API will be used
  - Replace: replace a received invalid value. The replacement value is the initvalue

Example


```
interface senderReceiver sri1 {
    data uint8 data1 queued initValue 15
    data uint32 data2
    data float32 data3
}
```

ClientServerInterface

- A client may initiate the execution of an operation by a server that supports the operation
- The server executes the operation and immediately provides the client with the result(synchronous operation call) or else the client checks for the completion by itself (asynchronous function call)
- A client may not connect to more than one server such that one specific operation call would be handled by multiple servers (n:1)
- It is not possible to pass a reference to a ClientServerOperation as an argument in another ClientServerOperation

ClientServerInterface

- In a ClientServerInterface, a client requests an operation that is carried out by the server. The client will be notified of the operation's completion either by asking or waiting for the server to acknowledge the completion
- Client needs to provide values for ArgumentDataPrototypes that are "in" or "inout"
- A component can be both client and server

Example


```
interface clientServer csi1{
    error error1 1
    error error2 2
    operation btkint possibleErrors error1{
        in uint16 myarg1 policy useVoid
    }
    operation btkbool possibleErrors error2{
        out ^boolean myarg2 policy useArgumentType
        out uint32 myarg3 policy useArgumentType
    }
}
```

ClientServerInterface

- ServerArgumentImplPolicyEnum
  - useArgugemtType: the argument type of the runnable entity is derived from the AutosarDataType of the ArgumentPrototype
  - useArrayBaseType: the argument type of the runnable entity is derived from the AutosarDataType of the elements of the array that corresponds to the ArgumentPrototype. This represents the base type of the array in C
  - useVoid: the argument type of the runnable entity is void

Modes and ModeSwitchInterfaces

- Mode requester: the component that asks for a mode change from the mode manager. Uses the SenderReceiverInterface.
- Mode manager: the component or BSW module that own the modeGroup and can change the mode requested by a mode requester through a SenderReceiverInterface. The mode manager is responsible for changing the current mode through ModeSwitchInterfaces
- Mode user: a component that is notified by the mode manager of the new mode through a ModeSwitchInterface

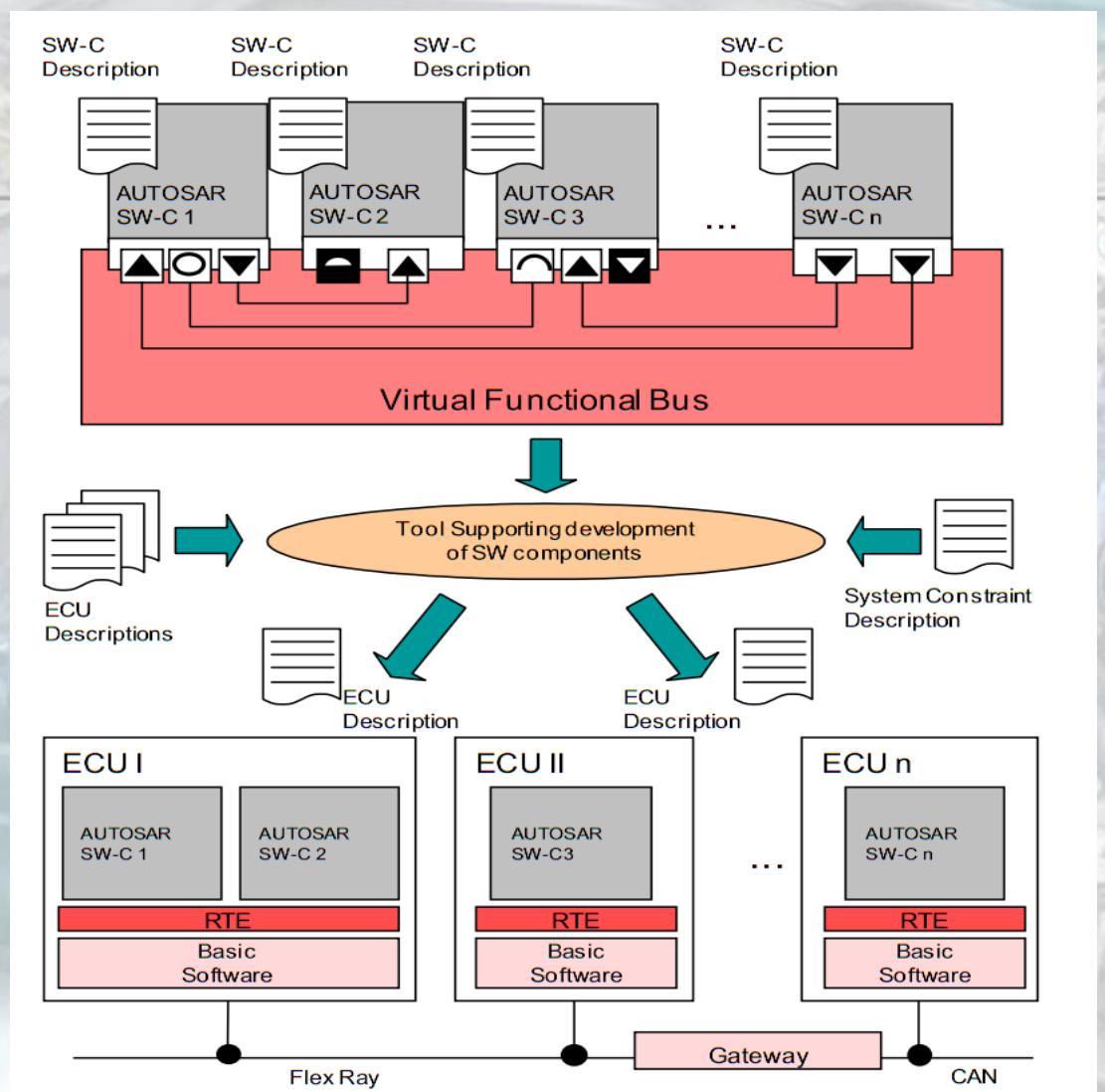
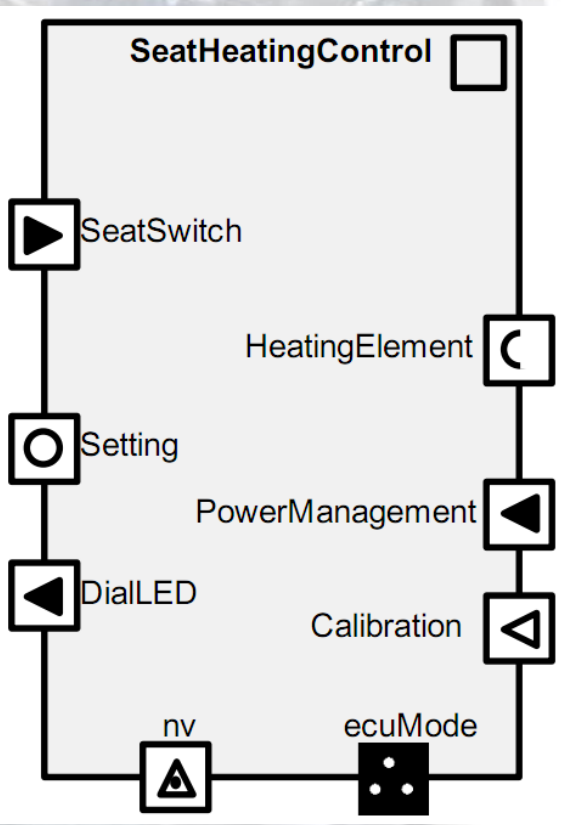


Figure 2.2: Detailed view on the activity "Configure System"

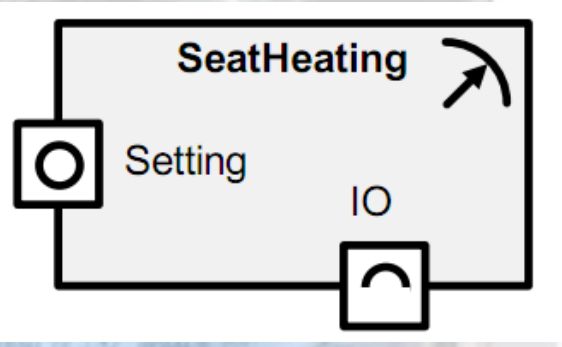
Example: Seat Heating Control

- Application SWC called SeatHeatingControl
- Ports:
  - Require ports:
    - If the seat is taken (SeatSwitch)
    - Setting of seat temperature dial(Setting)
    - Info from a power manager to decide when to turn off the heating (PowerManagement)
  - Provide ports:
    - dialled that is associated with the seat temperature dial (DialLED)
    - Heating element (HeatingElement)
  - The component can be calibrated (Calibration)
  - It needs the status of the ECU on which it runs (ecuMode)
  - Requires access to non-volatile memory (nv)

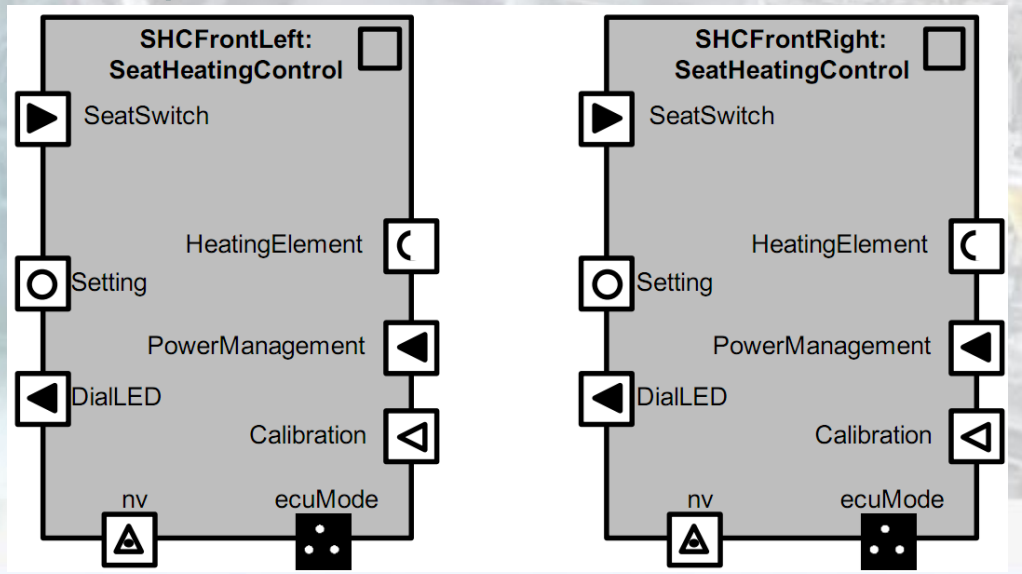


Example

- A SensorActuator SWC called SeatHeating
  - Inputs the desired setting of the heating element (Setting)
  - Directly controls the seat heating hardware (IO)

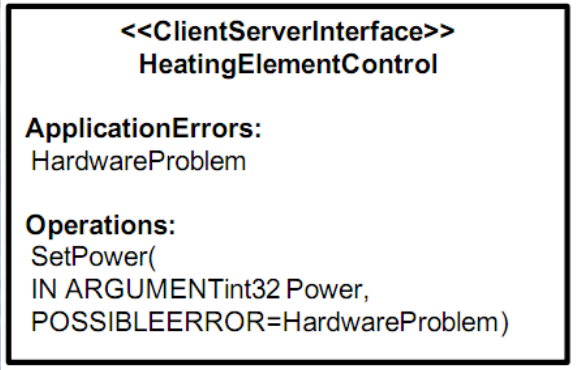


- AUTOSAR supports multiple instantiation

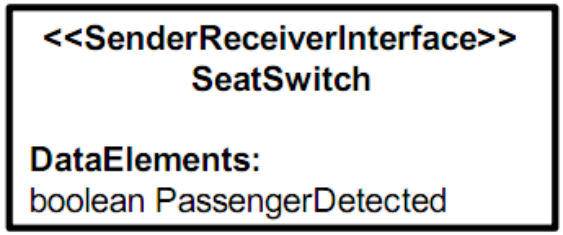


Example

- A ClientServerInterface defines a set of operations that can be invoked by a client and carried out by a server

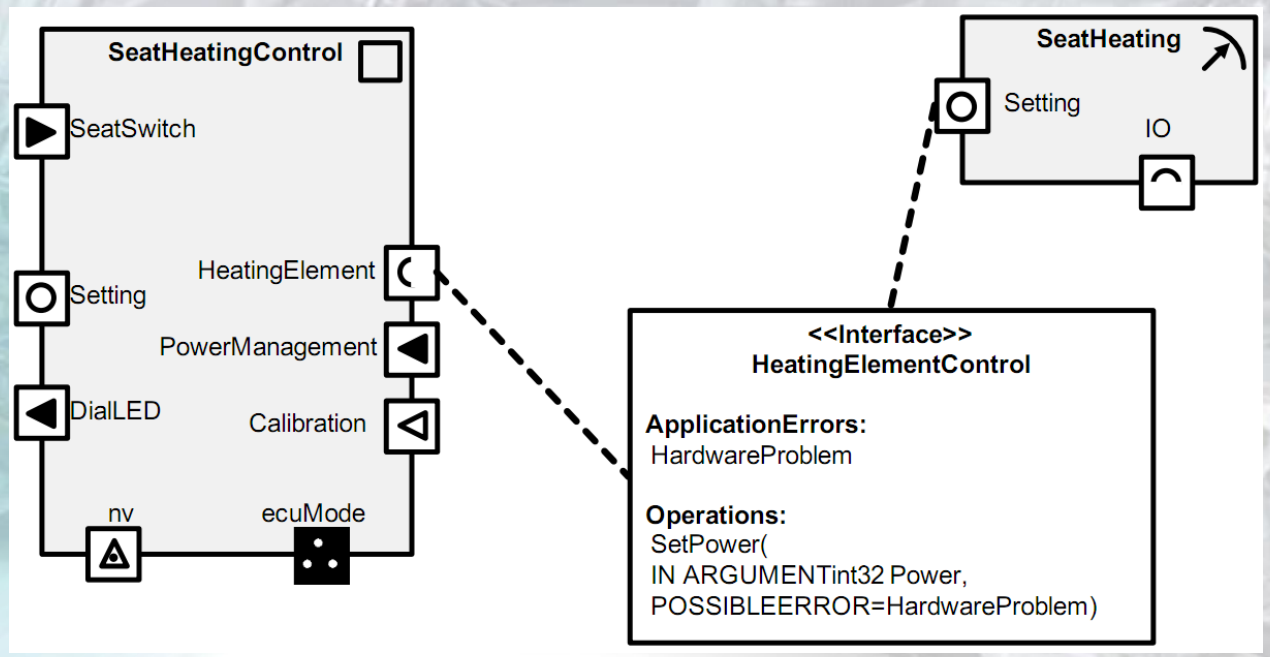


- A SenderReceiverInterface defines a set of data-elements that are sent and received



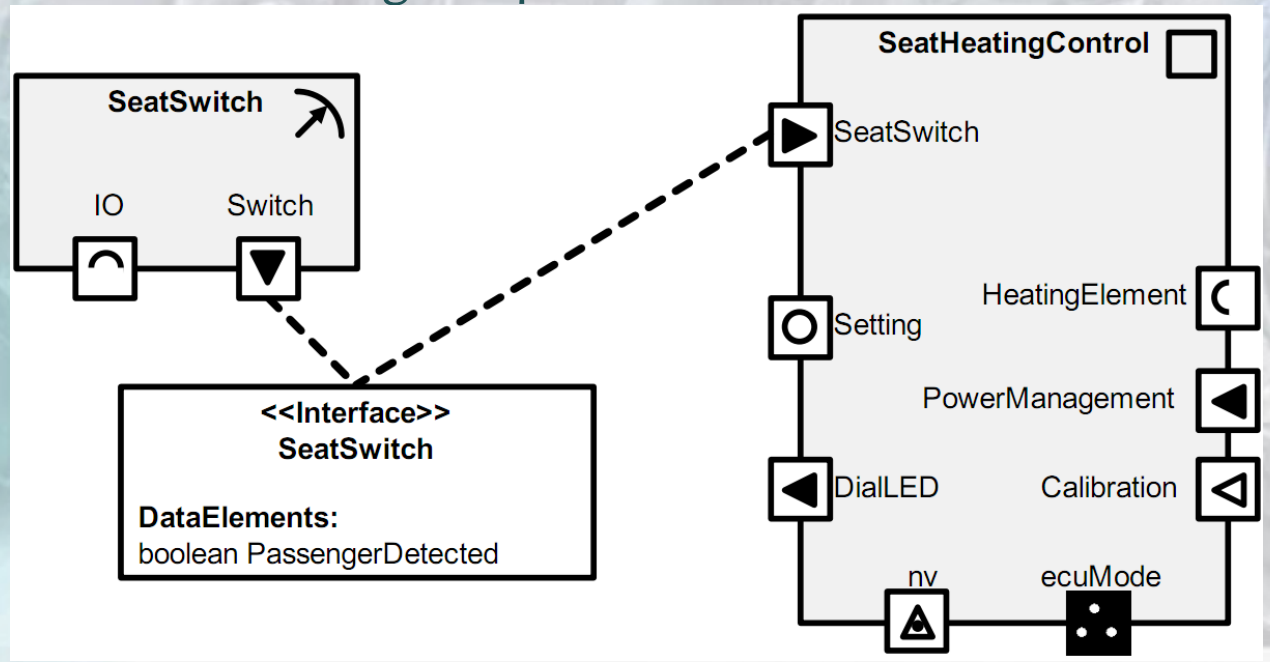
Example

- When a PPort provides a ClientServerInterface, the component to which the port belongs provides an implementation of the operations defined in the interface



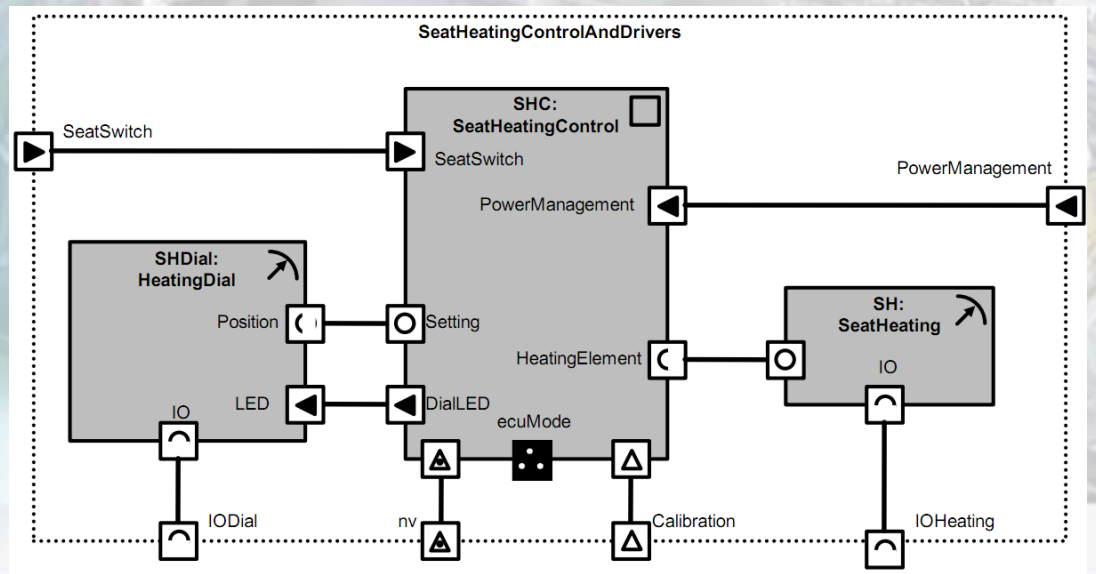
Example

- A component providing a SenderReceiverInterface generates values for the boolean value “PassengerDetected” through its port “Switch”. Similarly the component “SeatHeatingControl” can read the data-element “PassengerDetected” through its port “SeatSwitch”



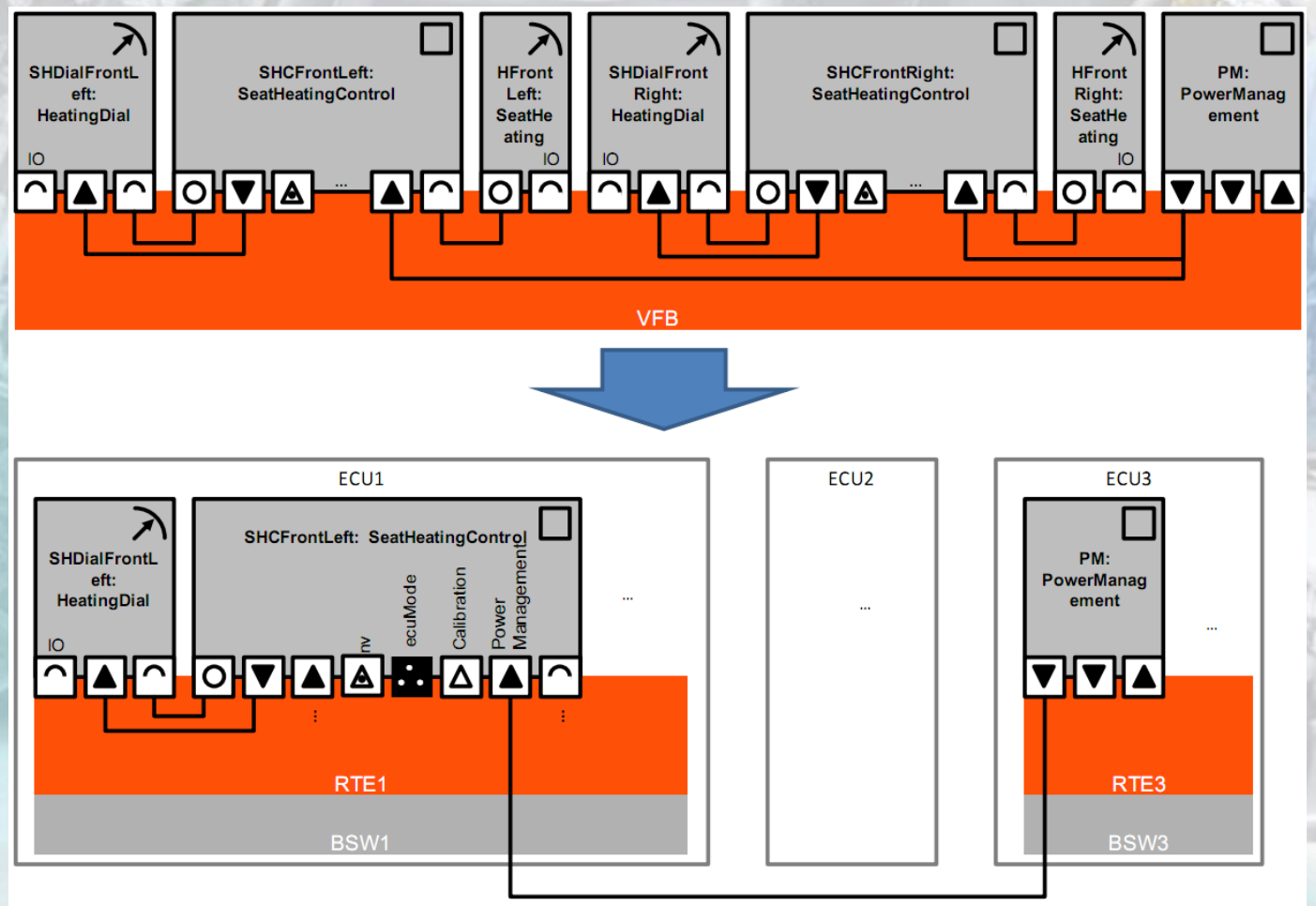
Example

- A sub-system consisting of usages of components and connectors is packaged into a composition. In AUTOSAR the usage of a component-type within a composition is called a prototype.
- This composition contains 3 prototypes: SHDial (type HeatingDial), SHC (type SeatHeatingControl), SH (type SeatHeating)

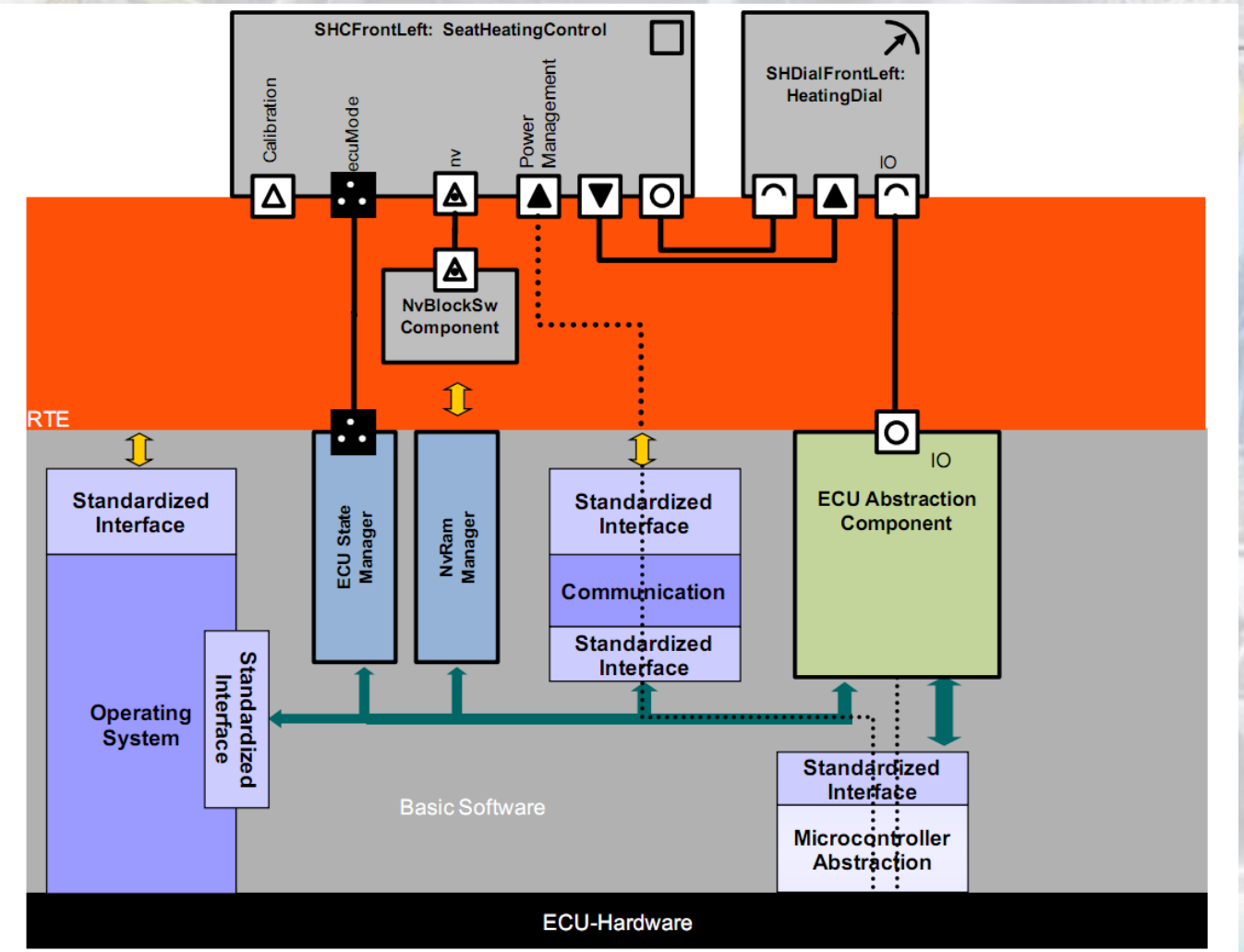


Example

- Mapping SWCs to ECUs



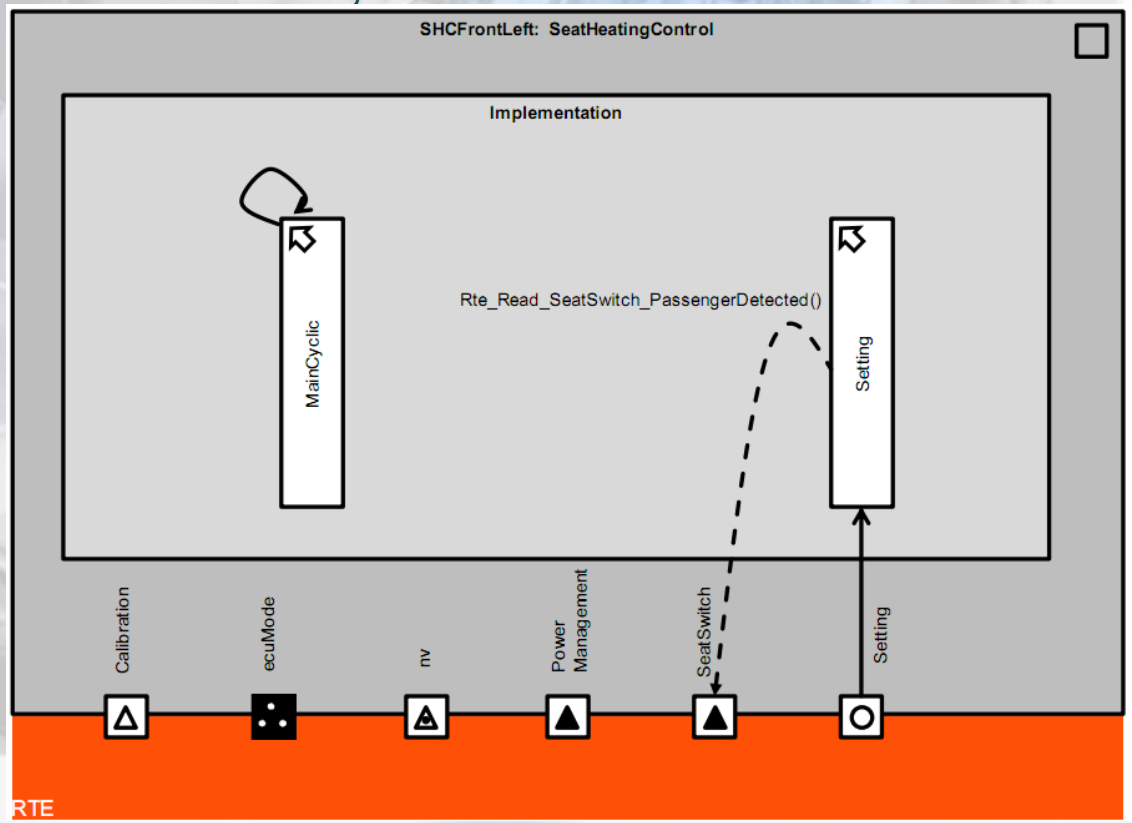
Example


SHCFrontLeft: SeatHeatingControl
Calibration
ecuMode
nv
Power Management
SHDialFrontLeft: HeatingDial
IO
NvBlockSw Component
RTE
Standardized Interface
ECU State Manager
NvRam Manager
Standardized Interface
Communication
Standardized Interface
IO
ECU Abstraction Component
Operating System
Standardized Interface
Basic Software
Standardized Interface
Microcontroller Abstraction
ECU-Hardware

Example

- Runnable entities
- A runnable entity is a sequence of instructions that can be started by the RTE
- A runnable runs in the context of a task (OS task)
- A task provides resources such as context

and stack size to a runnable



RTE Events

- asynchronousServerCallReturns: raised when an asynchronousservercall is finished
- dataReceiveErrorEvent: raised by RTE when the com layer detects and notifies an error regarding the received data is reported, references a variabledataprototype. Two cases where the com layer notifies the event:
  - The data value is equal to a predefined invalid value
  - The last update time of a periodic signal exceeds the alivetimeoutvalue
- dataReceivedEvent: raised when a referenced variabledataprototype is received
- dataSendCompletedEvent: raised when a sender has completed the transmission of the reference dataprototype or when an error is raised
- dataWriteCompletedEvent: raised when an implicit write access was successful or an error occurred

RTE Events

- modeSwitchEvent: raised when a mode change is received
- operationInvokedEvent: raised when an operation referenced by the interface is requested by the client
- Timingevent: raised periodically by RTE

Runnable Entity Attributes

- asynchronousServerCallResultPoint: the owning runnable entity is entitled to fetch the result of the asynchronous server call
- dataReadAccess: runnable entity has implicit read access to data element of a senderreceiver or nv portprototype
- dataReceivePoint: runnable entity has explicit read access to data element of a senderreceiver or nv portprototype
  - dataReceivePointByArgment: the result is passed back to the application by means of an argument in the function signature
  - dataReceivePointByValue: the result is passed back to the application by means of the return value
- dataSendPoint: runnable entity has explicit write access to dataElement of a senderreceiver or nv portprototype

Runnable Entity Attributes

- dataWriteAccess: runnable entity has implicit write access to data element of a senderreceiver or nv port
- Modeaccesspoint: a mode access point is required by a runnable entity owned by a mode manager or mode user. Its semantics implies the ability to access the current mode
- modeSwitchPoint: required by a runnable entity owned by a mode manager. Its semantics imply the ability to initiate a mode switch
- parameterAccess: the presence of a parameteraccess implies that a runnable entity needs read only access to a parameterdataprototype which may either be local or within a portprototype

Runnable Entity Attributes

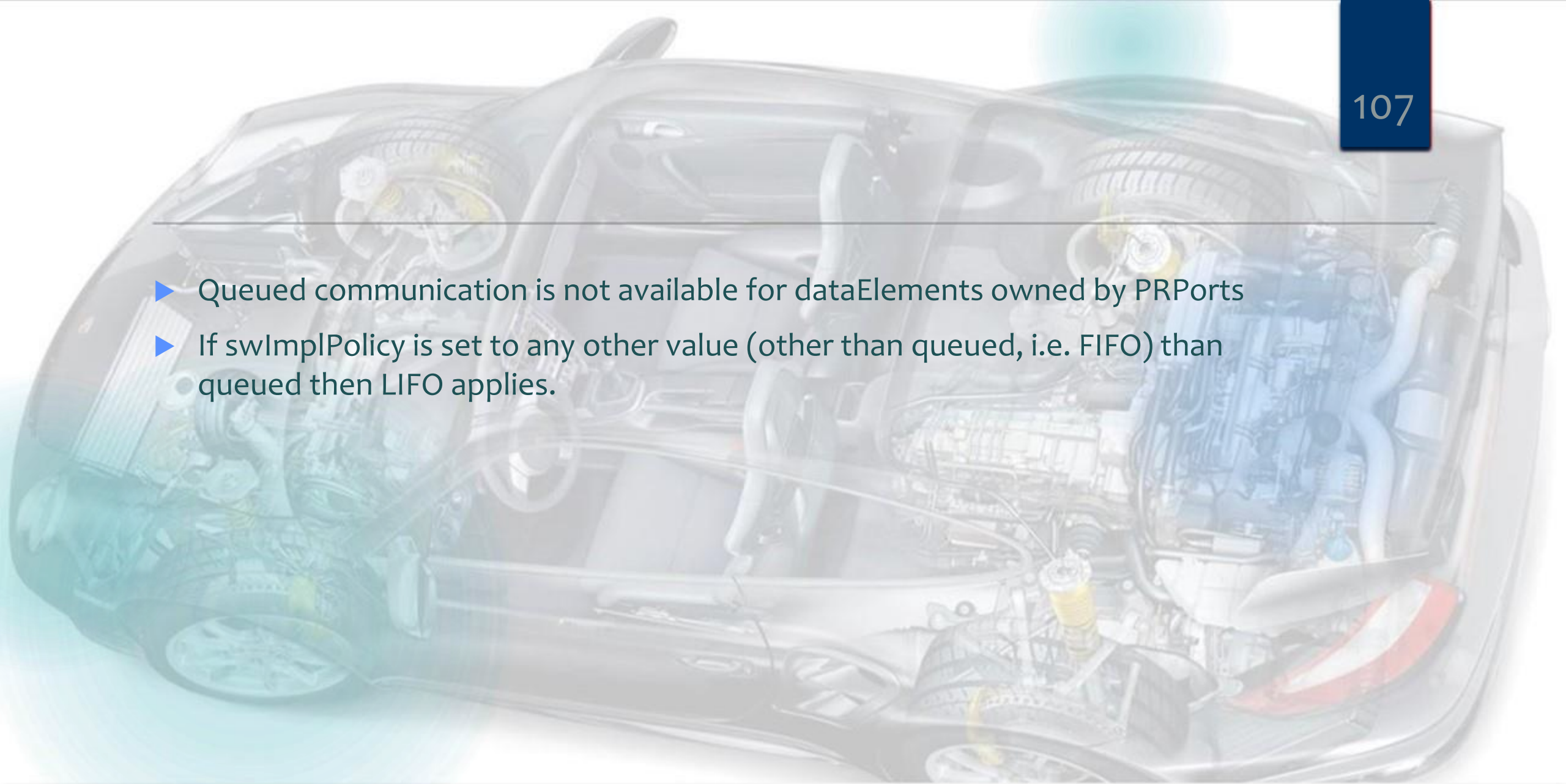
- serveCallPoint: if a runnable entity owns a server call point it is entitled to invoke a particular client server operation of a specific rportprototype of the corresponding atomicswcomponenttype
- Waitpoint: has a trigger that its waiting for
- externalTriggeringPoint: if a runnable entity owns it, it is entitled to raise an externaltriggeroccurredevent
- internalTriggeringPoint: if a runnable entity owns it, it is entitled to trigger the execution of runnable entities of the corresponding SWC
- The term implicit is used for communication based on data-access and explicit is used for data-point based communication

Measurement

- Only the following can be measured in AUTOSAR:
  - In the context of communication between SWCs:
    - VariableDataPrototypes enclosed in a SenderReceiverInterface
    - Argument of ClientServerOperations enclosed in a ClientServerInterface
  - In the context of a single SWC (internal):
    - Content of InterrunnableVariables which are used for communication between runnables of one AUTOSAR SWC

Timing

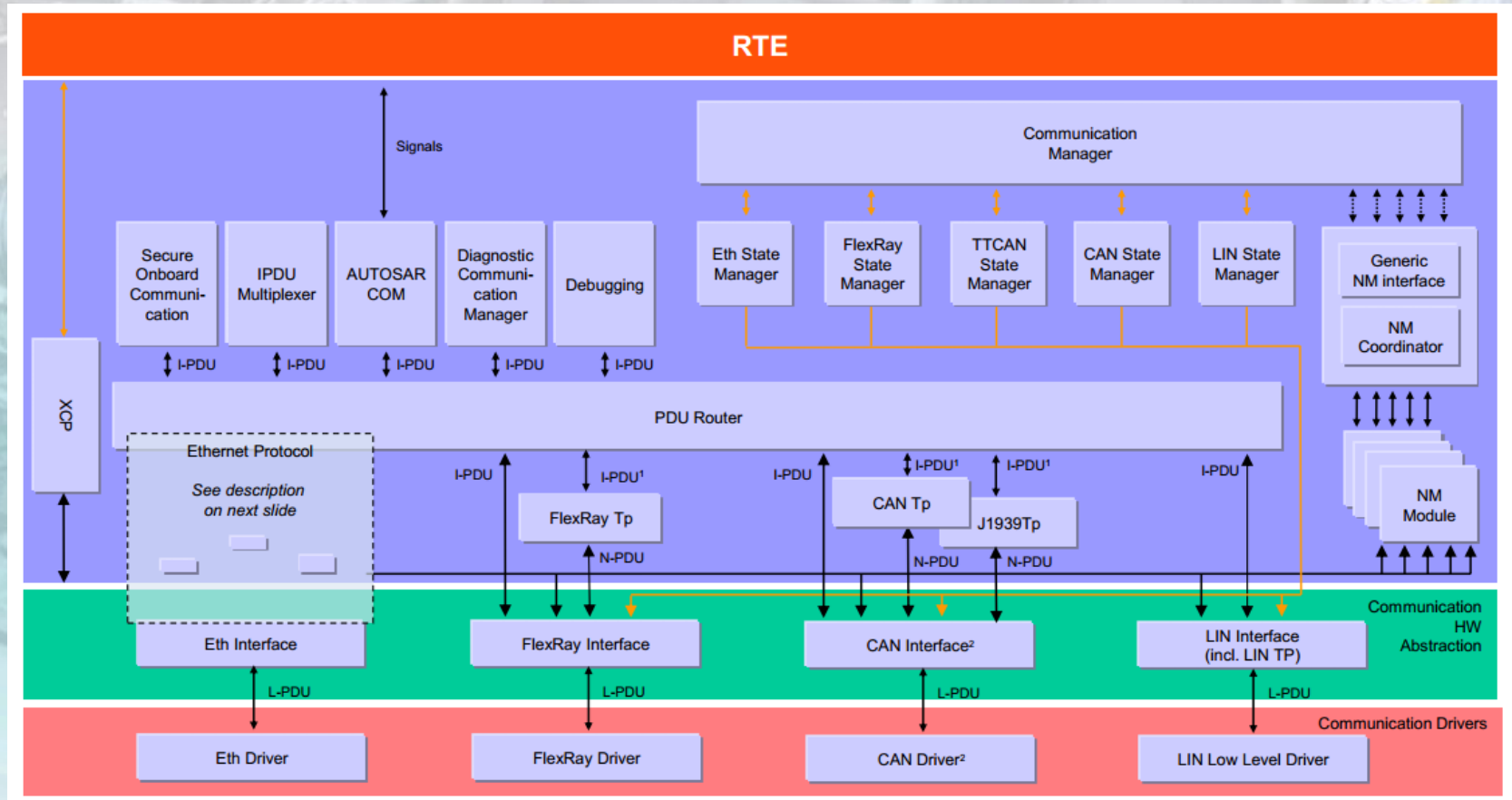
- VFBTiming: this view deals with timing information related to the interaction of SwComponentTypes at VFB level
- SwcTiming: this view deals with timing information related to the SwcInternalBehavior of AtomicSwComponentTypes
- SystemTiming: this view deals with timing information related to a system utilizing information about topology, software deployment and signal mapping
- BswModuleTiming: this view deals with timing information related to the BswInternalBehavior of a single BswModuleDescription
- EcuTiming: this view deals with timing information related to the EcuValueCollection, particularly with the EcuModuleConfigurationValues



- Queued communication is not available for dataElements owned by PRPorts
- If swImplPolicy is set to any other value (other than queued, i.e. FIFO) than queued then LIFO applies.

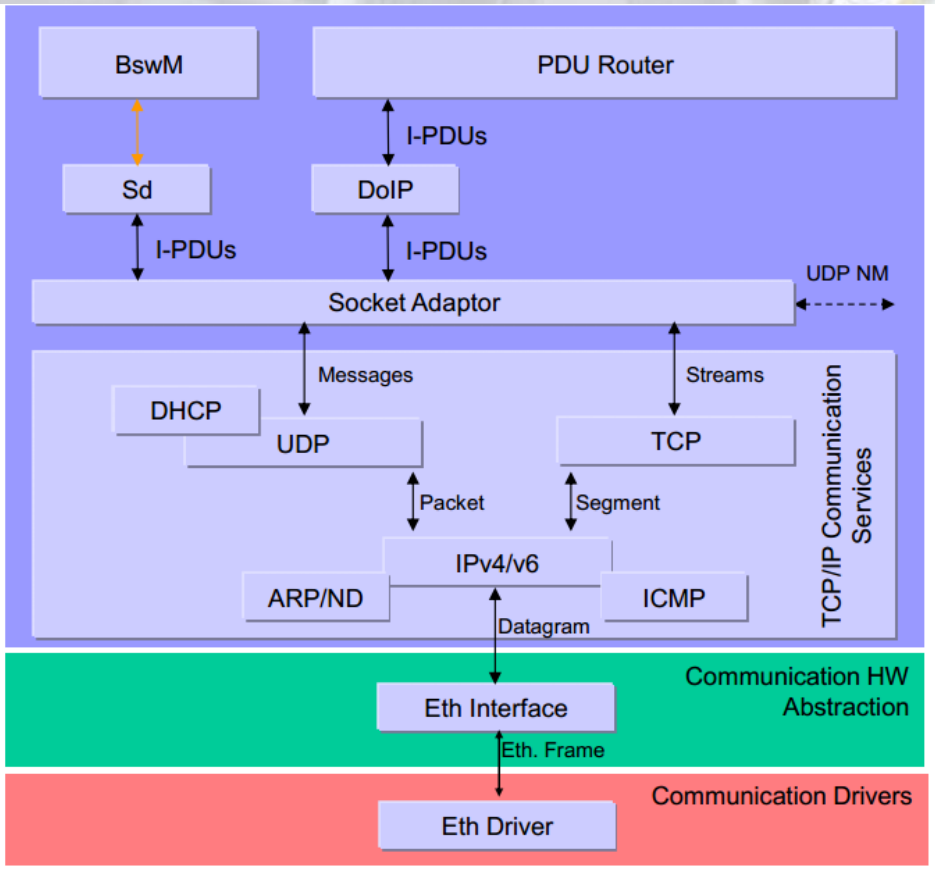
- Variant handling: it allows designers at many levels to put together a superset of functionality and choose which actual pieces of this functionality will be enabled in a specific variant
- AUTOSAR supports several discrete binding times:
  - System design
  - Code generation
  - Pre compile
  - Link time
  - Post build

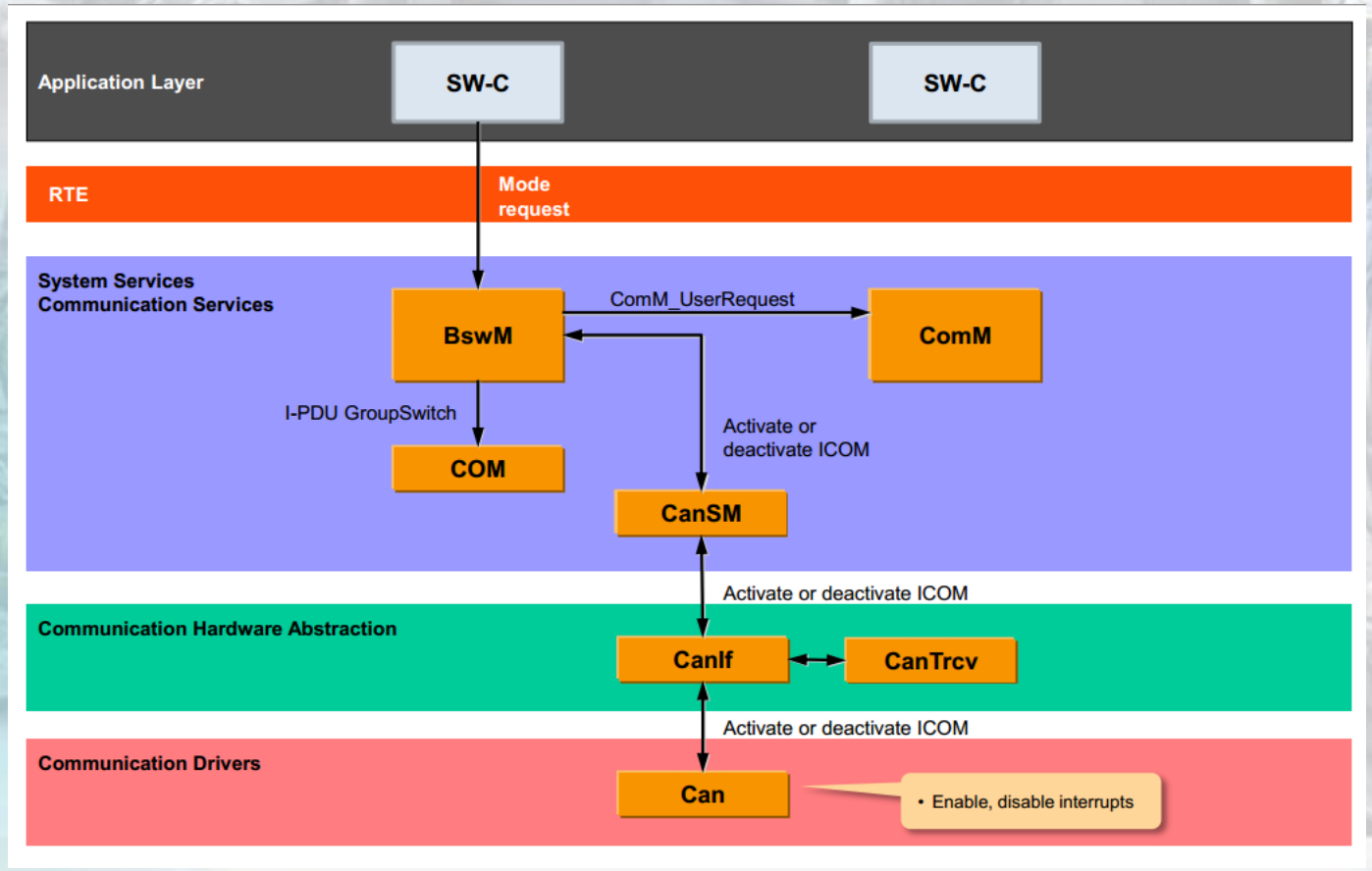
Interaction of Layers



Interaction of Layers

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




Application Layer
SW-C
SW-C
RTE
Mode request
System Services
Communication Services
BswM
ComM_UserRequest
ComM
I-PDU GroupSwitch
COM
Activate or deactivate ICOM
CanSM
Activate or deactivate ICOM
Communication Hardware Abstraction
CanIf
CanTrcv
Activate or deactivate ICOM
Communication Drivers
Can
• Enable, disable interrupts

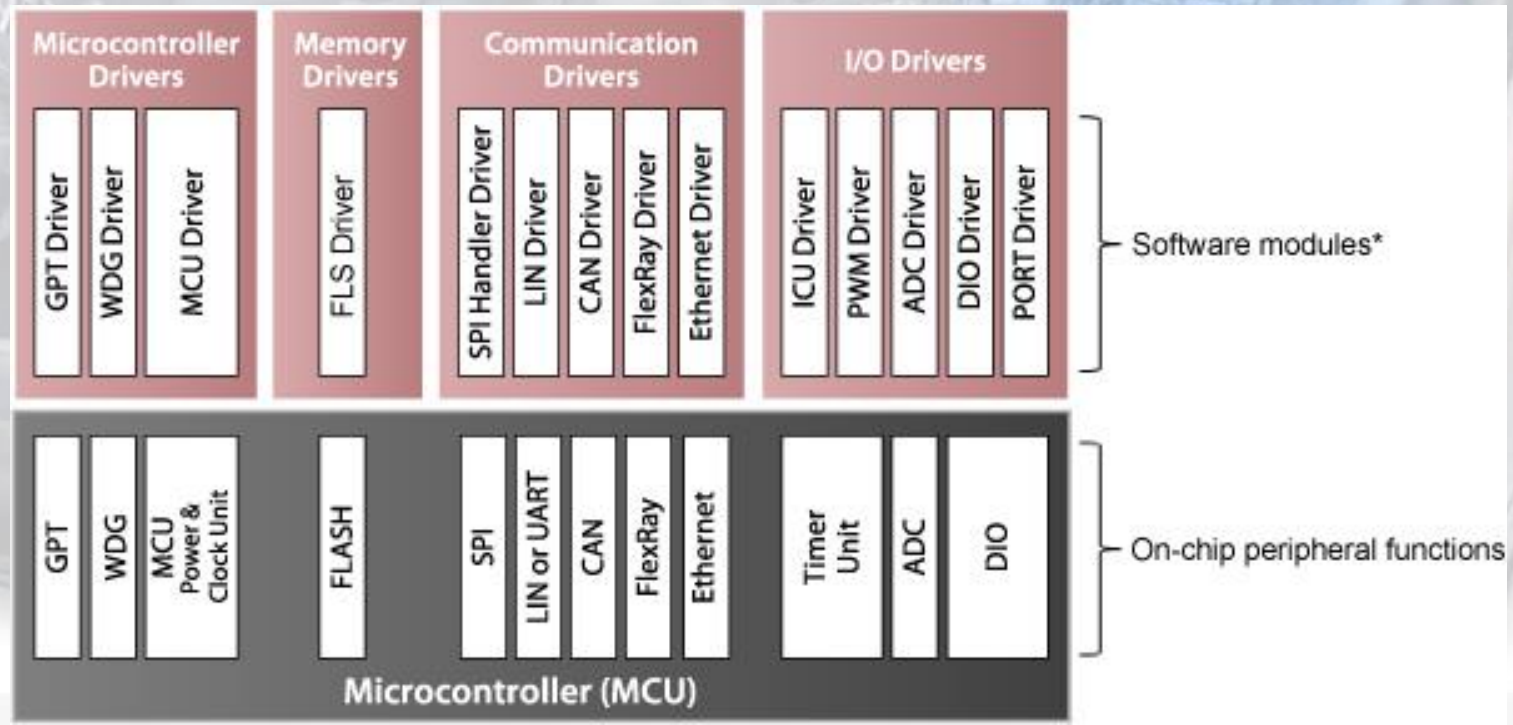
Autosar Interview Questions

- What is AUTOSAR?
- What is SWC?
- Difference between Intra ECU and Inter ECU Communication?
- What is meant by Client-Server Communication and Sender-Receiver Communication
- What is meant by Communication Stack?
- What is Pack and Unpacking IPdu?
- What is MDT(Minimum Delay Timer)?
- What is TMS (Transmission Mode Selection)?
- Explain about AUTOSAR COM module?
- What is RTE ? What are its function?
- How the SWC interact with CAN module?
- What is NM?
- What are functions of CANSM, CANIF & CAN module?
- Example of DET errors?
- Example of DEM errors?
- What is the functionality of DCM module
- Explain the AUTOSAR architecture?
- What are the pros & cons of AUTOSAR?
- What is meant by Pre-Compile, Post-Build & Link Tme

What is Next ?

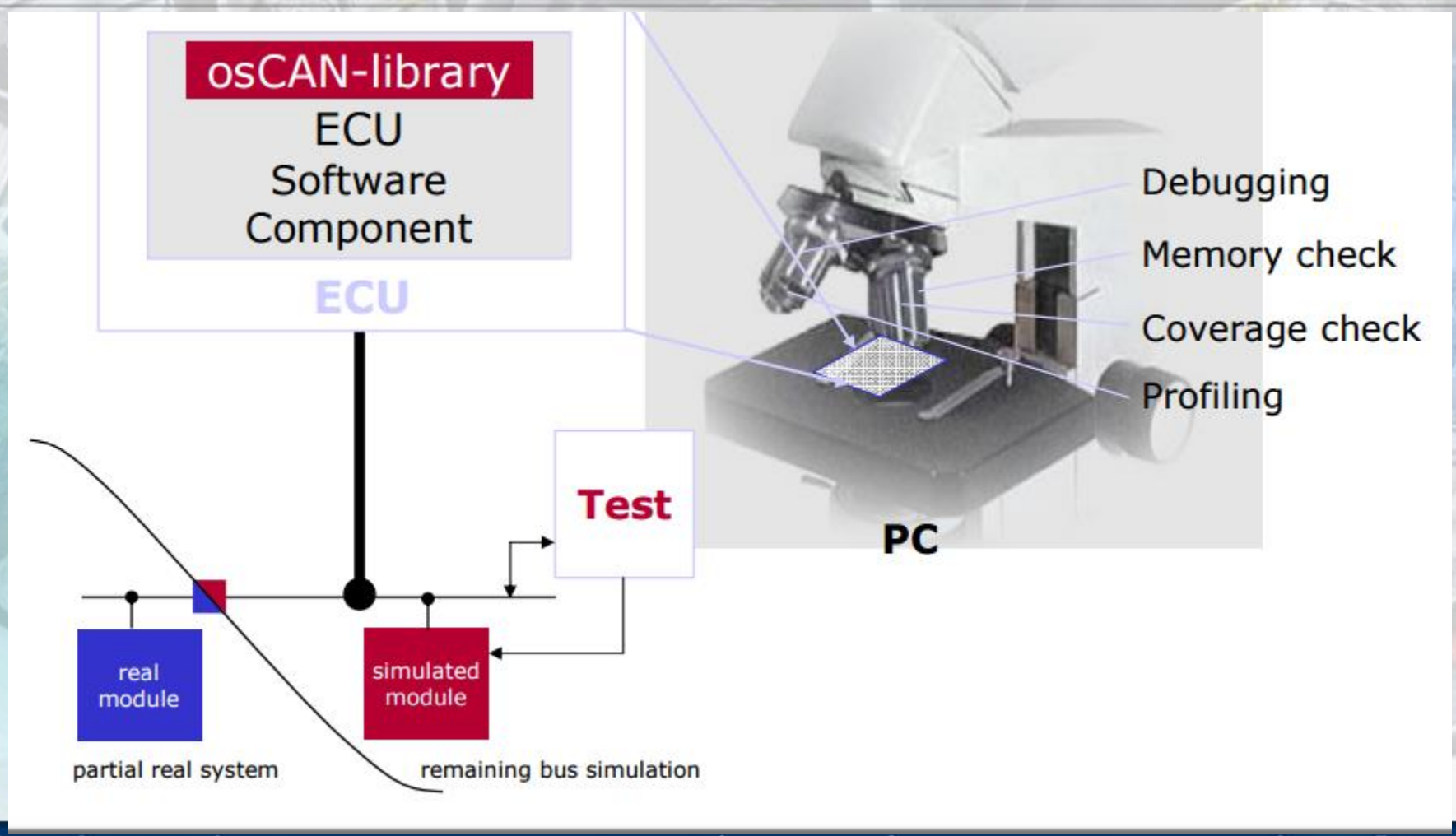
- We will create a MCAL "(Microcontroller Abstraction Layer)" Drivers For Atmega32 according to Autosar Specifications

MCAL (Microcontroller Abstraction Layer)
MCAL is a software module that directly accesses on-chip MCU peripheral modules and external devices that are mapped to memory, and makes the upper software layer independent of the MCU.
Details of the MCAL software module are shown below.



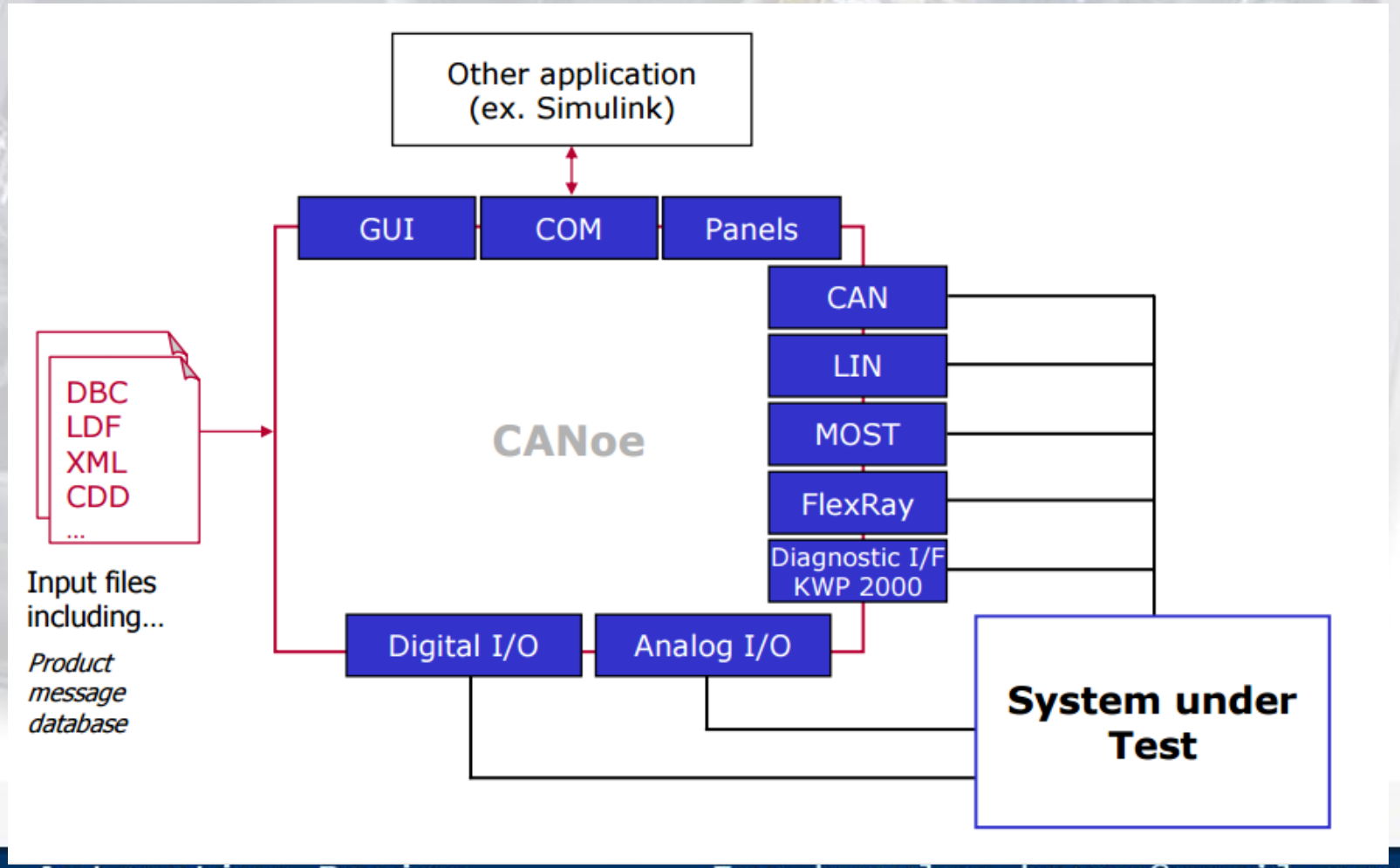
Automotive Modeling

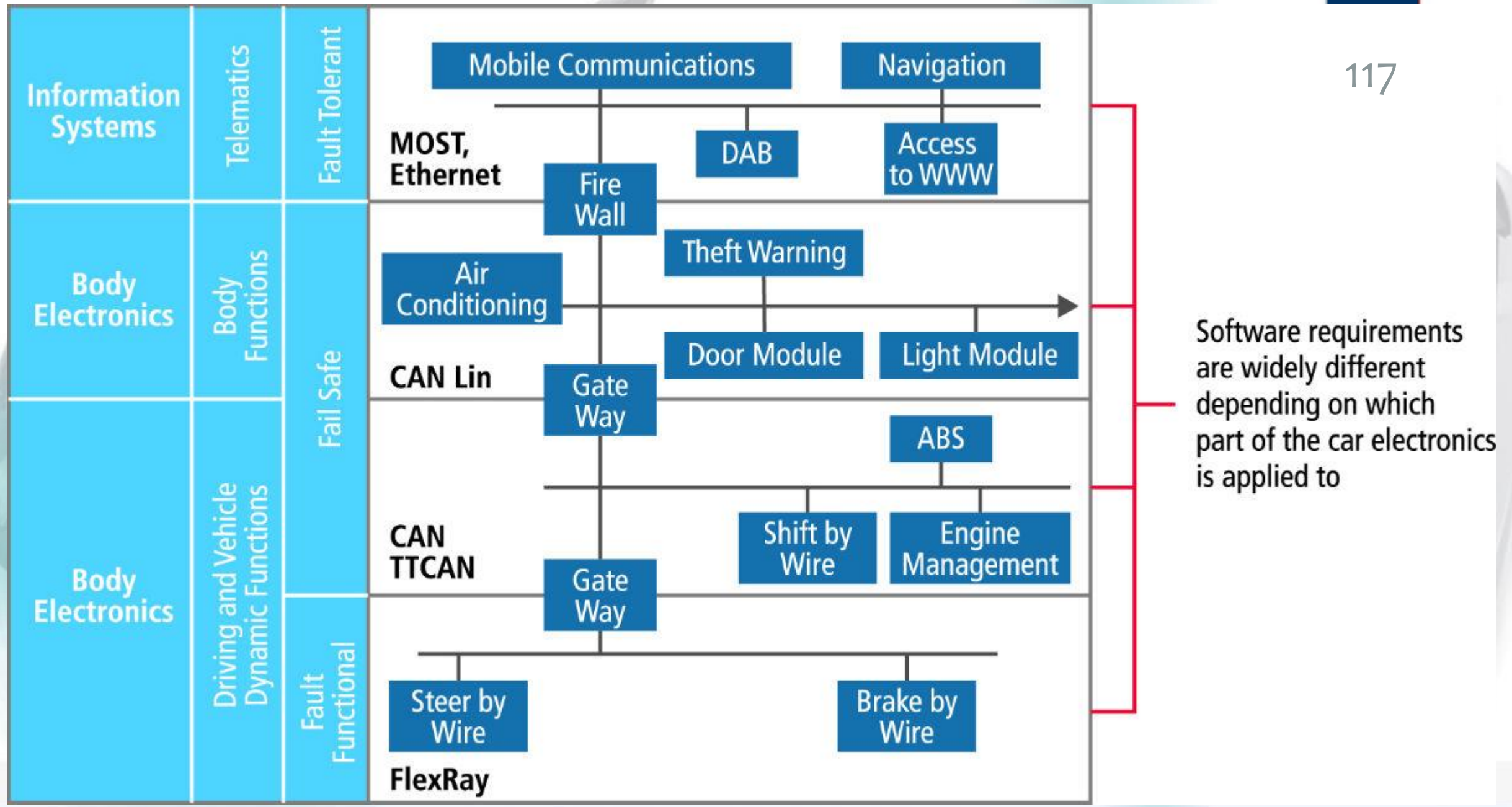
ECU Testing by Canoe



Vector CANoe

- CANoe is the comprehensive software tool for development, test and analysis of entire ECU networks and individual ECUs. It supports you throughout the entire development process - from planning up to final system-level tests.
- Its versatile options and functions are each providing matching project support. Therefore CANoe has been successfully in use at OEMs and suppliers for 20 years.

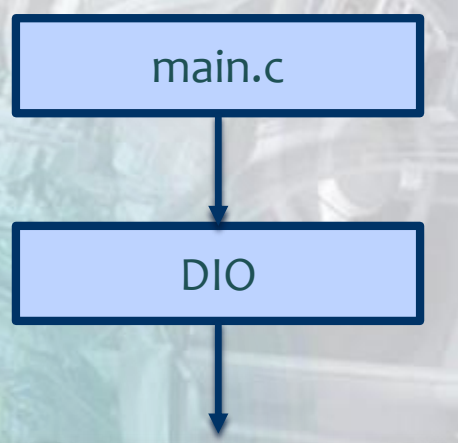



Information Systems
Telematics
Fault Tolerant
Body Electronics
Body Functions
Fail Safe
Body Electronics
Driving and Vehicle Dynamic Functions
Fault Functional
Mobile Communications
Navigation
DAB
Access to WWW
MOST, Ethernet
Fire Wall
Theft Warning
Air Conditioning
Door Module
Light Module
CAN Lin
Gate Way
ABS
Shift by Wire
Engine Management
CAN TTCAN
Gate Way
Steer by Wire
Brake by Wire
FlexRay
Software requirements are widely different depending on which part of the car electronics is applied to

Autosar Documentations Refrences

- AUTOSAR_EXP_LayeredSoftwareArchitecture
- Autosar_ppt
- 27239727-Automotive-Embedded-System-Development-in-AUTOSAR
- lesson19_autosar
- CommunicationStack_gosda

How to write DIO AYTOSAR MCAL for atmega32

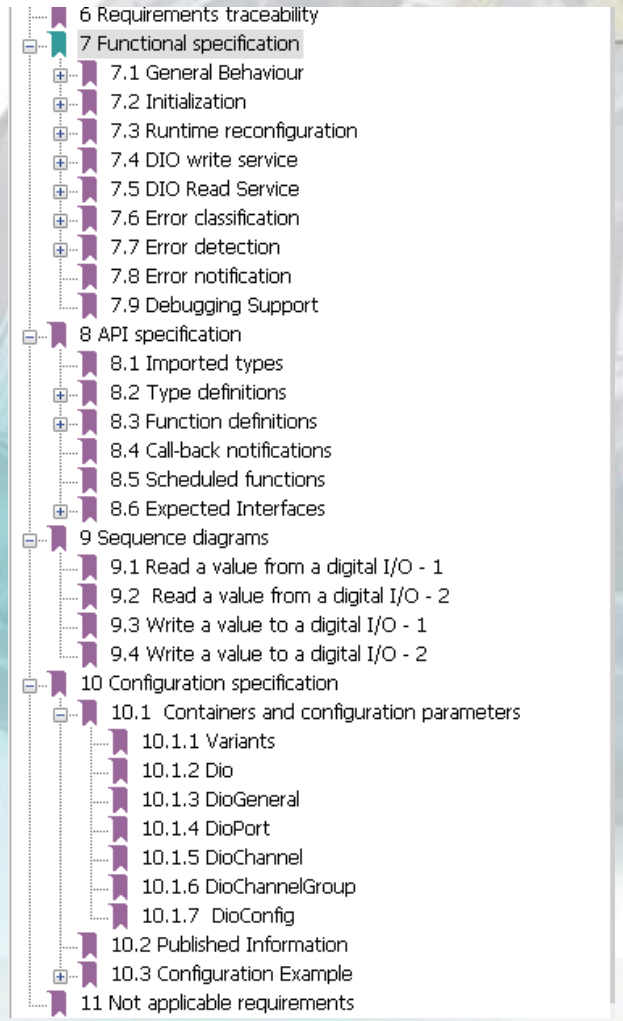


MCAL LAYER

Dio_Cfg.h | *Dio.h | Dio.c | main.c

```c
/*
 * main.c
 *
 *  Created on: Jan 19, 2017
 *      Author: Keroles Shenouda
 */



#include "drivers/DIO_AUTOSAR_MCAL/Dio.h"
#include <avr/delay.h>
//#include <avr/io.h>


int main (void)
{
    *DDRA= 0xff  ; // todo using PORT MCAL
    *DDRB = 0xff  ; // todo using PORT MCAL
    *DDRC = 0xff  ; // todo using PORT MCAL
    *DDRD = 0xff  ; // todo using PORT MCAL
    while (1)
    {
//      Dio_WritePort(DIO_PORT3 , 0xff) ; //PORTD_5
//      //_delay_ms(10);
//      Dio_WritePort(DIO_PORT3, 0x0) ; //PORTD_5


                Dio_WriteChannel (19 , 1) ; //PORTD_5
                //_delay_ms(10);
                Dio_WriteChannel(19, 0) ; //PORTD_5

    }



    return 0 ;
}
```

How to write DIO MCAL for atmega32



- Read First

  AUTOSAR_SWS_DIODriver.pdf from

  Autosar.org

DIO Driver Structure and Integration

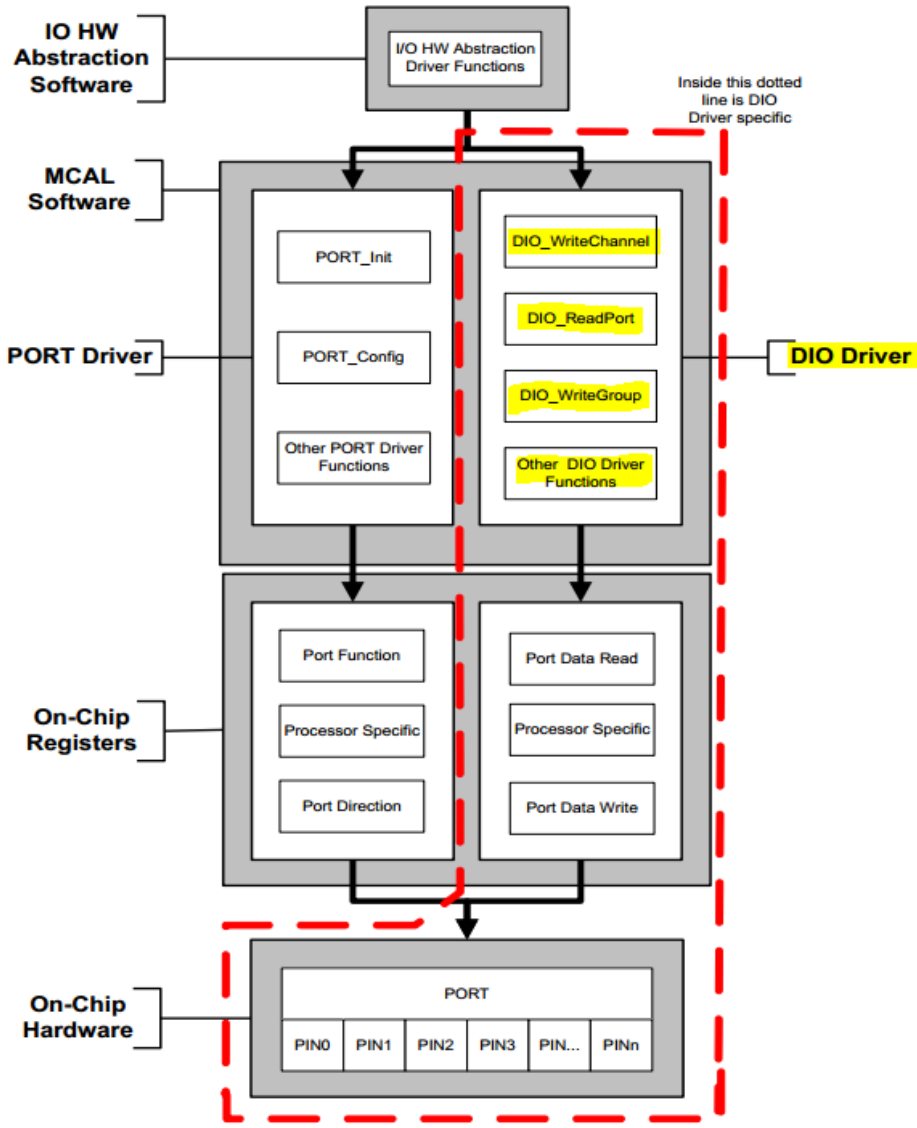


Figure 1: DIO Driver Structure and Integration

Dependencies to other modules

Port Driver Module

Many ports and port pins are assigned by the PORT Driver Module to various functionalities as for example:

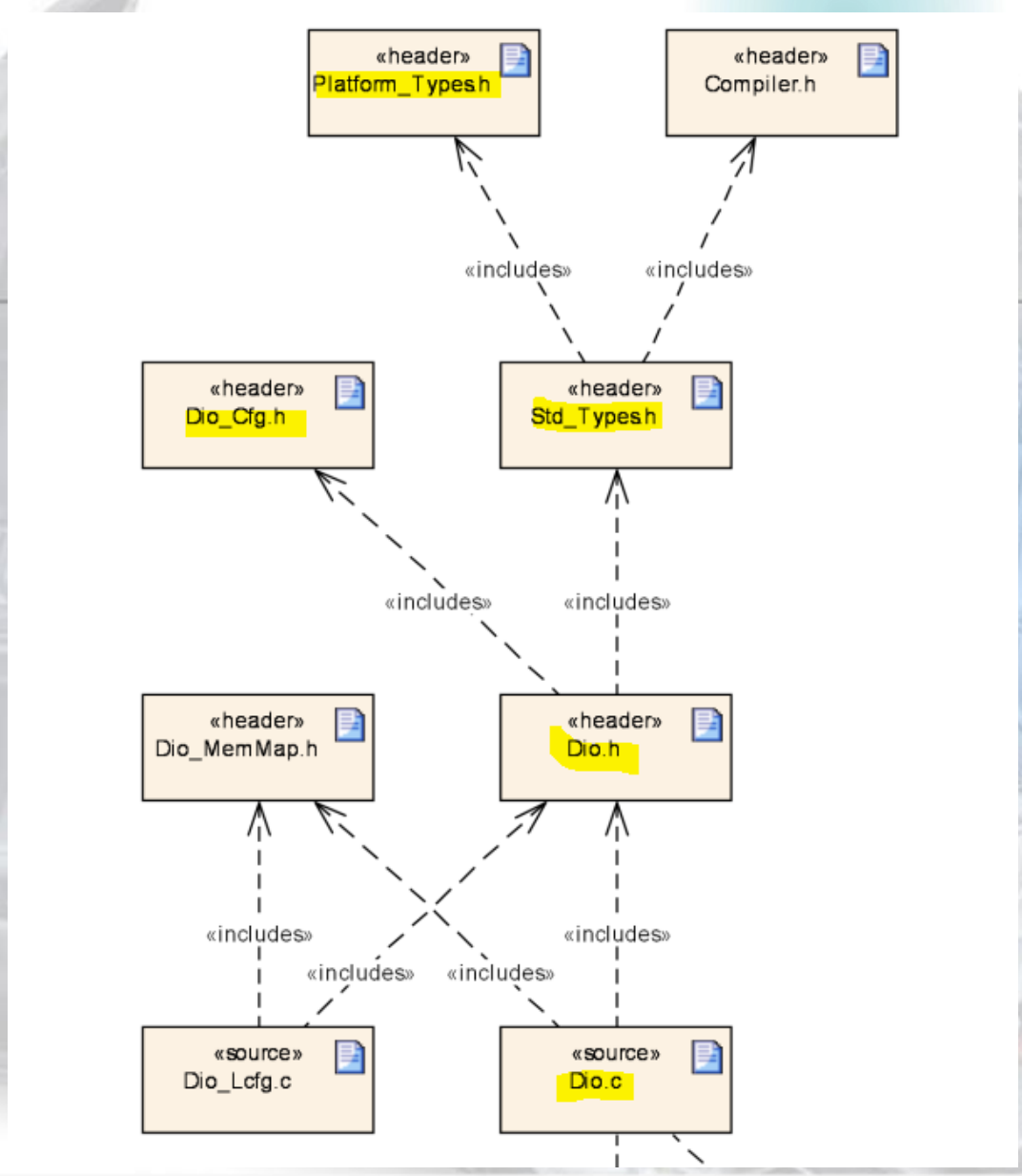
- General purpose I/O
- ADC
- SPI
- PWM

[SWS_Dio_00061] ⌈The Dio module shall not provide APIs for overall configuration and initialization of the port structure which is used in the Dio module. These actions are done by the PORT Driver Module.⌋ ()

[SWS_Dio_00063] ⌈The Dio module shall adapt its configuration and usage to the microcontroller and ECU.⌋ ()

[SWS_Dio_00102] ⌈The Dio module's user shall only use the Dio functions after the Port Driver has been initialized. Otherwise the Dio module will exhibit undefined behavior.⌋ ()

File structure



API service ID's

DIO_prog.c | DIO_private.h | DIO_interface.h | DIO_config.h | Platform_Types.h | Std_Types.h

```
/*
 * Dio.h
 *
 *  Created on: Jan 20, 2017
 *      Author: Keroles shenouda
 */

#ifndef DEBUG_DRIVERS_DIO_AUTOSAR_MCAL_DIO_H_
#define DEBUG_DRIVERS_DIO_AUTOSAR_MCAL_DIO_H_


//=================================================

#include "Std_Types.h"
//=================================================



//=================================================

// API Service ID's
//=================
//  Service name                Service ID[hex]
#define DIO_READCHANNEL_ID          0x00
#define DIO_WRITECHANNEL_ID         0x01
#define DIO_READPORT_ID             0x02
#define DIO_WRITEPORT_ID            0x03
#define DIO_READCHANNELGROUP_ID     0x04
#define DIO_WRITECHANNELGROUP_ID    0x05
#define DIO_GETVERSIONINFO_ID       0x12
#define  Dio_FlipChanne_ID          0x11
//=================================================


```

- API—覧

API	ID [Dec]	ID [Hex]
Dio_ReadChannel	0	0x00
Dio_WriteChannel	1	0x01
Dio_ReadPort	2	0x02
Dio_WritePort	3	0x03
Dio_ReadChannelGroup	4	0x04
Dio_WriteChannelGroup	5	0x05
Dio_GetVersionInfo	18	0x12
Dio_Init	16	0x10
Dio_FlipChannel	17	0x11

8.3.1 Dio_ReadChannel

[SWS_Dio_00133] [

Service name:	Dio_ReadChannel	
Syntax:	`Dio_LevelType Dio_ReadChannel( Dio_ChannelType ChannelId )`	
Service ID[hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ChannelId	ID of DIO channel
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Dio_LevelType	STD_HIGH The physical level of the corresponding Pin is STD_HIGH STD_LOW The physical level of the corresponding Pin is STD_LOW
Description:	Returns the value of the specified DIO channel.	

] ()

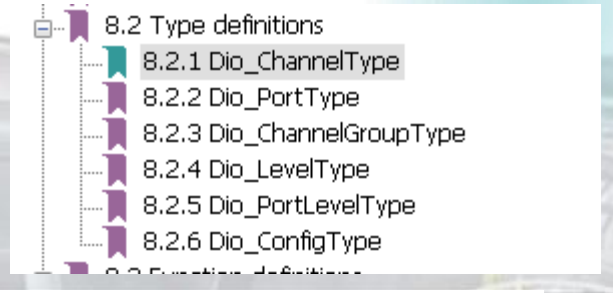
Error classification

7.6.1 Development Errors

Type of error	*Relevance*	*Related error code*	*Value [hex]*
[SWS_DIO_00175] ⌈Invalid channel name requested⌋ ()	Development	DIO_E_PARAM_INVALID_CHANNEL_ID	0x0A
[SWS_DIO_00176] ⌈API service called with "NULL pointer" parameter⌋ ()	Development	DIO_E_PARAM_CONFIG	0x10
[SWS_DIO_00177] ⌈Invalid port name requested⌋ ( )	Development	DIO_E_PARAM_INVALID_PORT_ID	0x14
[SWS_DIO_00178] ⌈Invalid ChannelGroup passed⌋ ()	Development	DIO_E_PARAM_INVALID_GROUP	0x1F
[SWS_DIO_00188] ⌈API service called with a NULL pointer. In case of this error,	Development	DIO_E_PARAM_POINTER	0x20

```
34
35  // Error classification
36  //=================
37  //  Service name                    Service ID[hex]
38  #define DIO_E_PARAM_INVALID_CHANNEL_ID      0x0A
39  #define DIO_E_PARAM_CONFIG                  0x10
40  #define DIO_E_PARAM_INVALID_PORT_ID         0x14
41  #define DIO_E_PARAM_INVALID_GROUP_ID        0x1F
42  #define DIO_E_PARAM_POINTER                 0x20
43  //=====================================================
```

Type definitions




```
43 //==================================
44
45 //Type definitions
46 //=================
47
48 typedef uint8 Dio_ChannelType;
49 typedef uint8 Dio_PortType;
50 typedef struct
51 {
52   Dio_PortType port;
53   uint8 offset;
54   uint8 mask;
55 } Dio_ChannelGroupType;
56 typedef uint8 Dio_LevelType;
57 typedef uint8 Dio_PortLevelType;
58
59 //==================================
```

8.2.1 Dio_ChannelType

[SWS_Dio_00182] ⌈

Name:	Dio_ChannelType		
Type:	uint		
Range:	This is implementation specific but not all values may be valid within the type.	--	Shall cover all available DIO channels
Description:	Numeric ID of a DIO channel.		

⌋ ()

[SWS_Dio_00015] ⌈Parameters of type `Dio_ChannelType` contain the numeric ID of a DIO channel. ⌋ ()

[SWS_Dio_00180] ⌈The mapping of the ID is implementation specific but not configurable. ⌋ ()

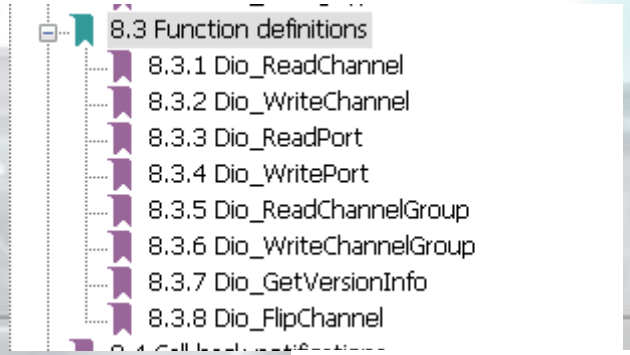
[SWS_Dio_00017] ⌈For parameter values of type `Dio_ChannelType`, the Dio's user shall use the symbolic names provided by the configuration description.

Furthermore, SWS_Dio_00103 applies to the type `Dio_ChannelType`. ⌋ (SRS_SPAL_12263, SRS_Dio_12355)

Version Number


```c
//Version Number
//=================
#define DIO_MODULE_ID               MODULE_ID_DIO
#define DIO_VENDOR_ID               VENDOR_ID_Keroles

#define DIO_SW_MAJOR_VERSION    1
#define DIO_SW_MINOR_VERSION    0
#define DIO_SW_PATCH_VERSION    0

#define DIO_AR_MAJOR_VERSION    4
#define DIO_AR_MINOR_VERSION    3
#define DIO_AR_PATCH_VERSION    0

//================================================
```

Function Prototypes



8.3.1 Dio_ReadChannel

[SWS_Dio_00133] [

Service name:	Dio_ReadChannel	
Syntax:	`Dio_LevelType Dio_ReadChannel( Dio_ChannelType ChannelId )`	
Service ID[hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ChannelId	ID of DIO channel
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Dio_LevelType	STD_HIGH The physical level of the corresponding Pin is STD_HIGH STD_LOW The physical level of the corresponding Pin is STD_LOW
Description:	Returns the value of the specified DIO channel.	

] ()

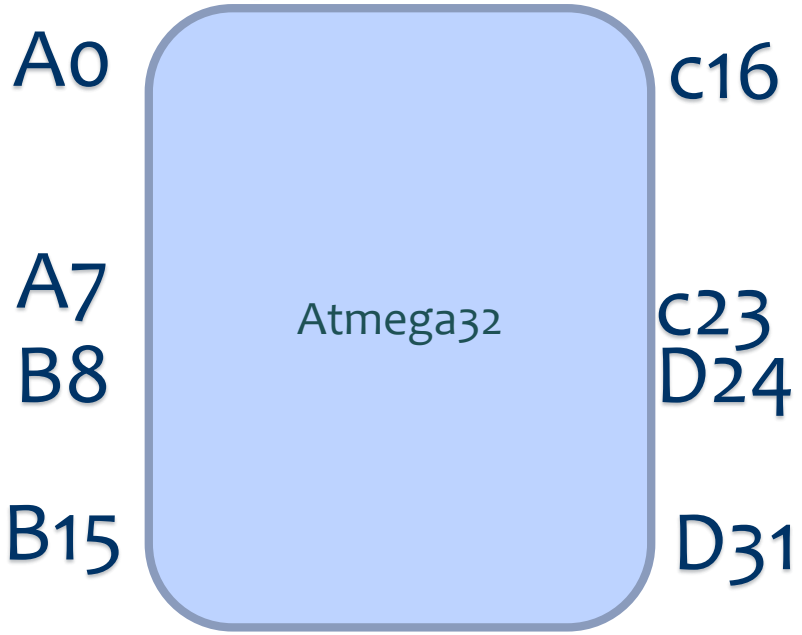
```
82 //Function Prototypes
83 //==================
84 .
85 Dio_LevelType Dio_ReadChannel(Dio_ChannelType channelId);
86 void Dio_WriteChannel(Dio_ChannelType channelId, Dio_LevelType level);
87
88 Dio_PortLevelType Dio_ReadPort(Dio_PortType portId);
89 void Dio_WritePort(Dio_PortType portId, Dio_PortLevelType level);
90
91 Dio_PortLevelType Dio_ReadChannelGroup( const Dio_ChannelGroupType *channelGroupIdPtr );
92 void Dio_WriteChannelGroup(const Dio_ChannelGroupType *channelGroupIdPtr, Dio_PortLevelType level);
93
94 #if ( DIO_VERSION_INFO_API == STD_ON)
95 #define Dio_GetVersionInfo(_vi) STD_GET_VERSION_INFO(_vi,DIO)
96 #endif
97 //=================================================
```

Dio_Cfg.h

Dio_Cfg.h

```
/*
 * Dio_Cfg.h
 *
 *  Created on: Jan 20, 2017
 *      Author: Keroles shenouda
 */

#ifndef DEBUG_DRIVERS_DIO_AUTOSAR_MCAL_DIO_CFG_H_
#define DEBUG_DRIVERS_DIO_AUTOSAR_MCAL_DIO_CFG_H_

#define DIO_ID_0          0
#define DIO_ID_1          1
#define DIO_ID_2          2
#define DIO_ID_3          3
#define DIO_ID_4          4
#define DIO_ID_5          5
#define DIO_ID_6          6
#define DIO_ID_7          7
#define DIO_ID_8          8
#define DIO_ID_9          9
#define DIO_ID_10        10
#define DIO_ID_11        11
#define DIO_ID_12        12
#define DIO_ID_13        13
#define DIO_ID_14        14
#define DIO_ID_15        15
#define DIO_ID_16        16
#define DIO_ID_17        17
#define DIO_ID_18        18
#define DIO_ID_19        19
#define DIO_ID_20        20
#define DIO_ID_21        21
```

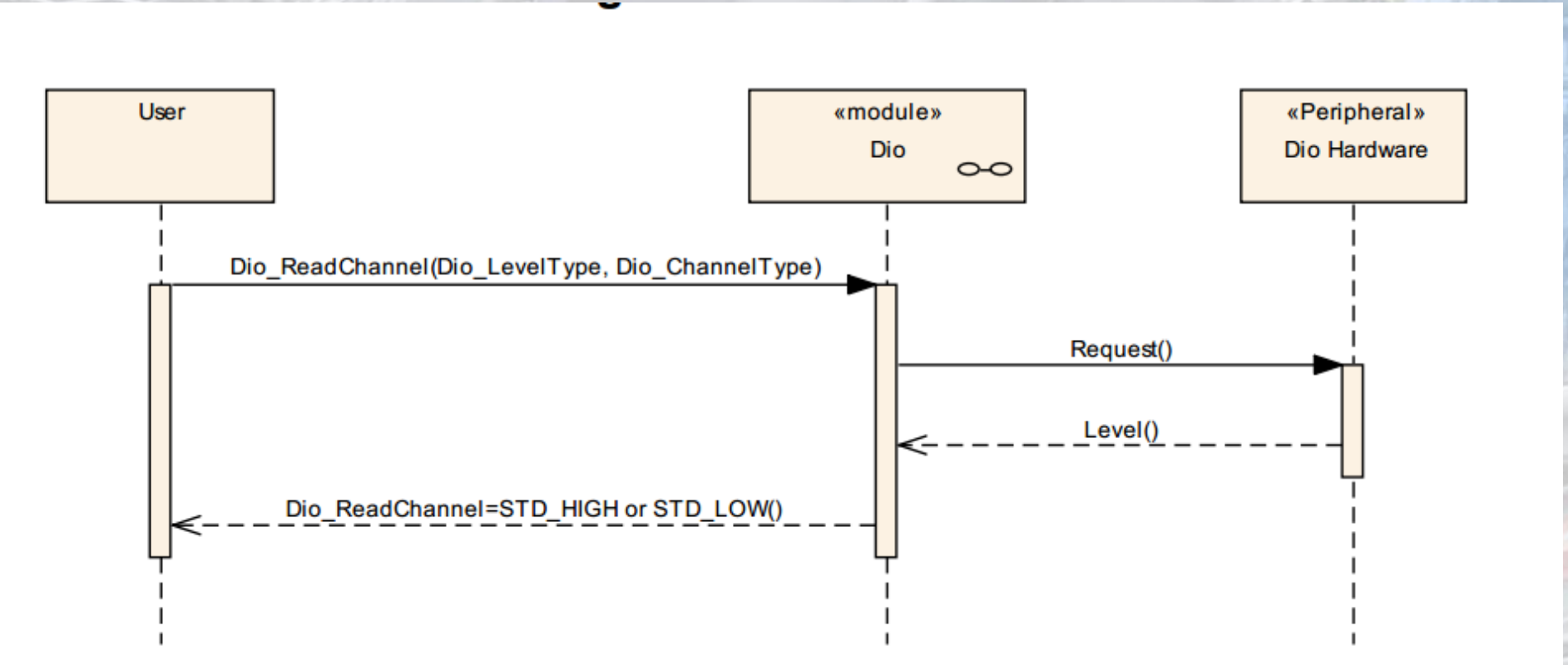
Assuming the Pins Mapped as :



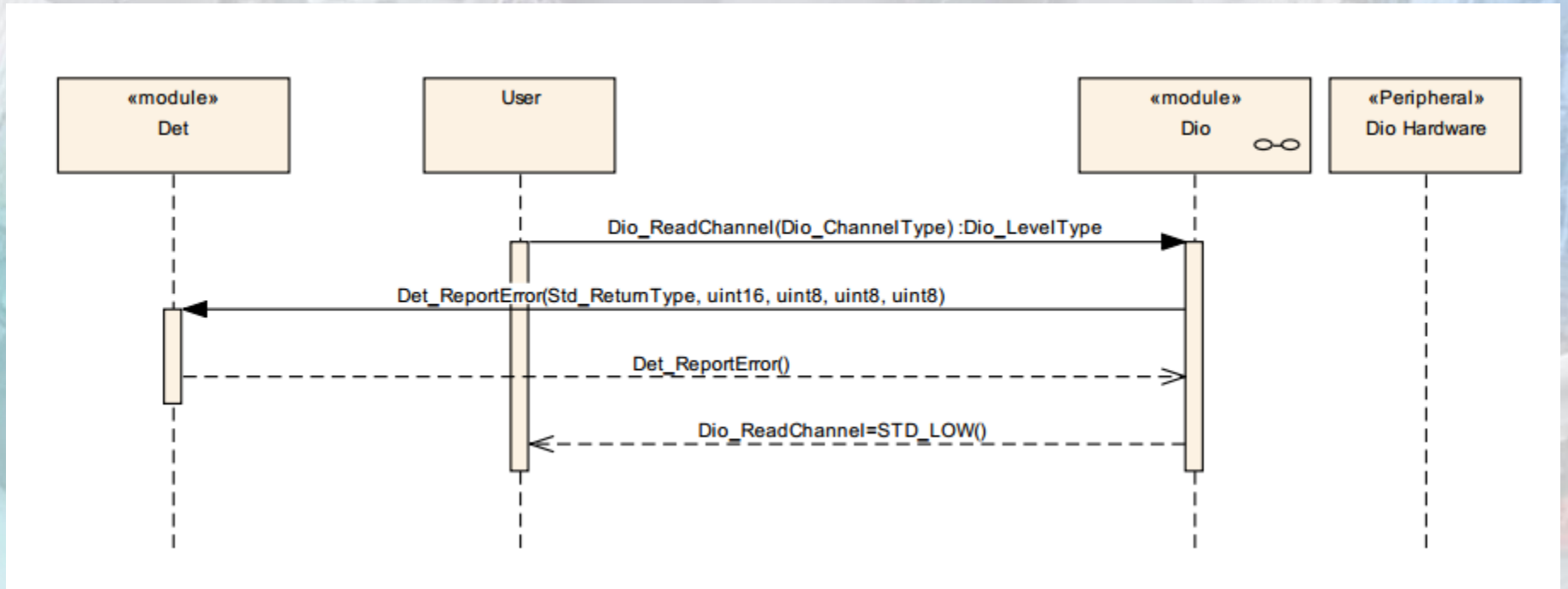
Outline | Make Target | Task List

- DEBUG_DRIVERS_DIO_AUTOSAR_MCAL_DIO_CF
- DIO_ID_0
- DIO_ID_1
- DIO_ID_2
- DIO_ID_3
- DIO_ID_4
- DIO_ID_5
- DIO_ID_6
- DIO_ID_7
- DIO_ID_8
- DIO_ID_9
- DIO_ID_10
- DIO_ID_11
- DIO_ID_12
- DIO_ID_13
- DIO_ID_14
- DIO_ID_15
- DIO_ID_16
- DIO_ID_17
- DIO_ID_18
- DIO_ID_19
- DIO_ID_20
- DIO_ID_21
- DIO_ID_22
- DIO_ID_23
- DIO_ID_24
- DIO_ID_25
- DIO_ID_26
- DIO_ID_27
- DIO_ID_28
- DIO_ID_29
- DIO_ID_30
- DIO_ID_31
- DIO_PORT0
- DIO_PORT1
- DIO_PORT2
- DIO_PORT3
- no_of_ports
- no_of_total_pins

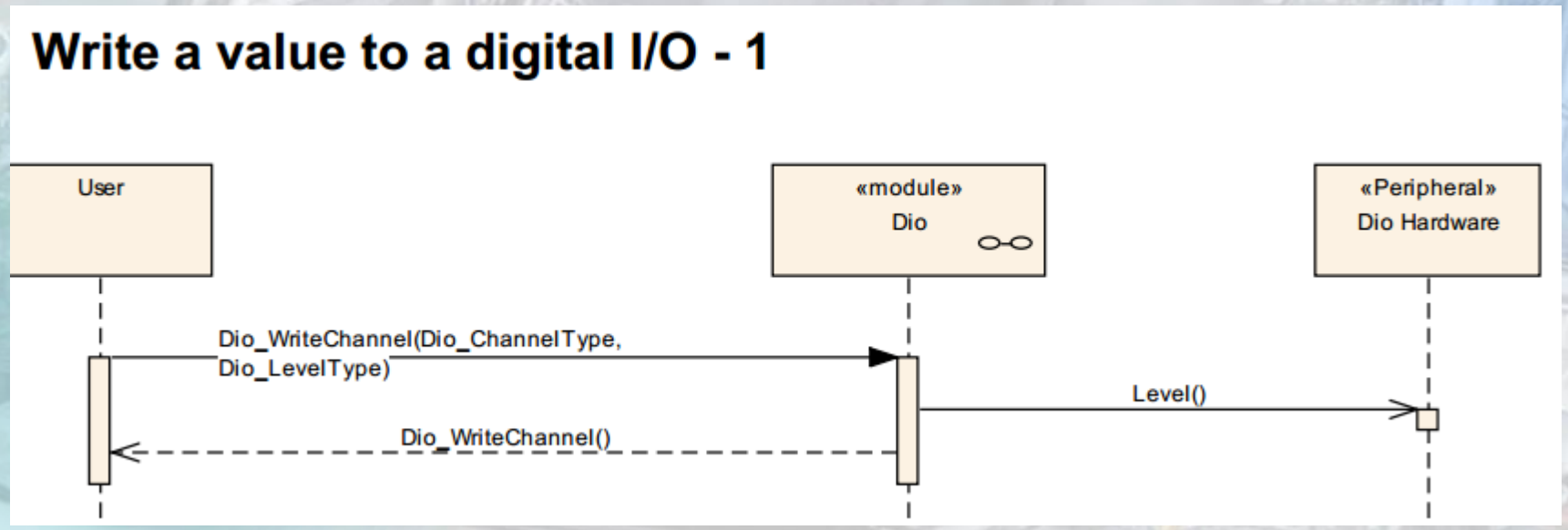
Dio_WriteChannel()
Dio_ReadChannel()



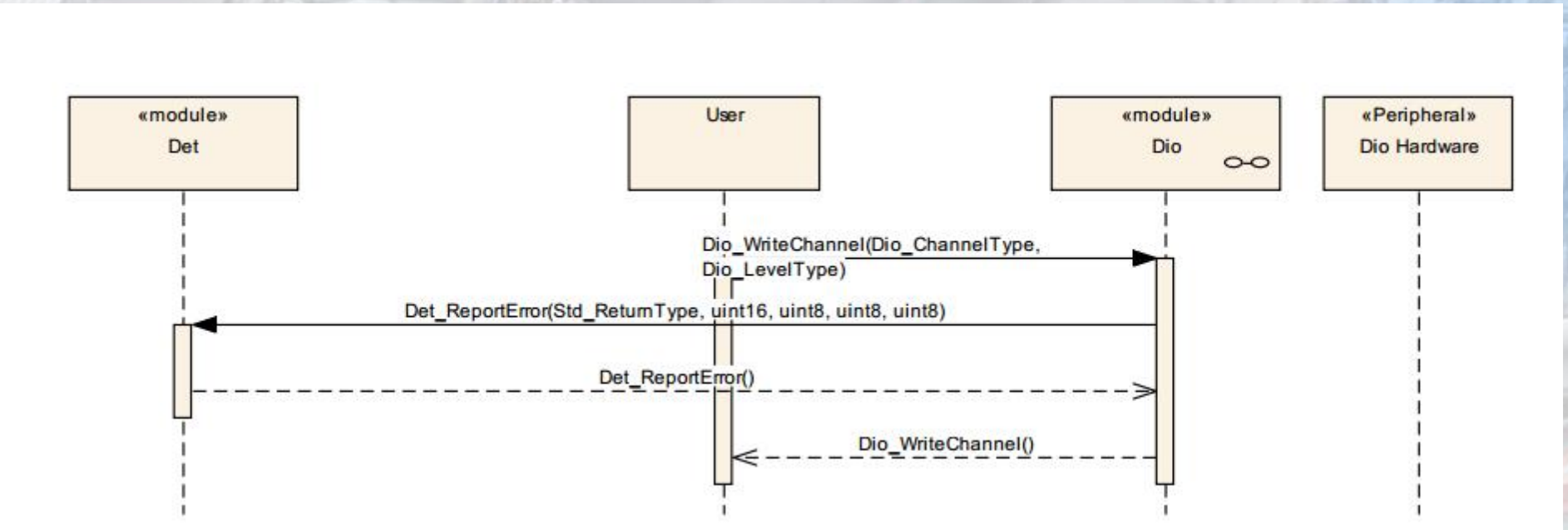
Dio_WriteChannel()
Dio_ReadChannel()



Dio_WriteChannel()
Dio_ReadChannel()



Dio_WriteChannel()
Dio_ReadChannel()



Dio_WriteChannel()
Dio_ReadChannel()

```
15  Dio_LevelType Dio_ReadChannel(Dio_ChannelType channelId)
16
17  {
18
19      Dio_LevelType  result ;
20
21      if(channelId > no_of_total_pins)
22      {
23          //todo check channel id, if not valid Report_error(DIO_E_PARAM_INVALID_CHANNEL_ID)
24      }
25      else
26      {
27          result  = GetBit(*(DIO_u8pins[(u8)(port_number)]),(channelId%no_of_pins));
28
29      }
30
31      if (result)
32          return STD_HIGH ;
33      else
34          return STD_LOW;
35  }
```

Dio_WriteChannel()
Dio_ReadChannel()

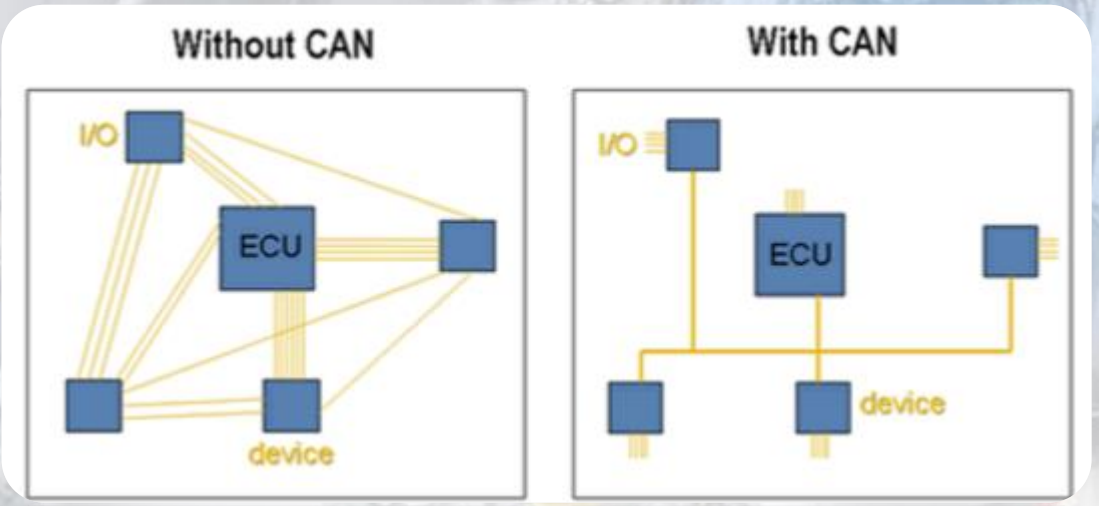

```
37 void Dio_WriteChannel(Dio_ChannelType channelId, Dio_LevelType level)
38 {
39
40     if(channelId > no_of_total_pins)
41     {
42         //todo check channel id, if not valid Report_error(DIO_E_PARAM_INVALID_CHANNEL_ID)
43     }
44     else
45     {
46         switch(level)
47         {
48         case STD_HIGH:
49             SetBit(*(DIO_u8ports[(u8)(port_number)]),(channelId%no_of_pins));
50
51             break;
52         case STD_LOW:
53             ClrBit(*(DIO_u8ports[(u8)(port_number)]),(channelId%no_of_pins));
54
55             break;
56         default:
57             break;
58
59         }
60     }
61
62 }
63
```

```
65 Dio_PortLevelType Dio_ReadPort(Dio_PortType portId)
66 {
67     Dio_PortLevelType result;
68
69     if(portId > no_of_ports)
70     {
71         //todo check channel id, if not valid Report_error(DIO_E_PARAM_INVALID_CHANNEL_ID)
72     }
73     else
74     {
75         result = *(DIO_u8pins[portId]);
76     }
77
78     return result ;
79 }
80
81 void Dio_WritePort(Dio_PortType portId, Dio_PortLevelType level)
82 {
83
84     if(portId > no_of_ports)
85     {
86         //todo check channel id, if not valid Report_error(DIO_E_PARAM_INVALID_CHANNEL_ID)
87     }
88     else
89     {
90         *(DIO_u8ports[portId]) = level;
91     }
92 }
93
```

```
99  Dio_PortLevelType Dio_ReadChannelGroup( const Dio_ChannelGroupType *channelGroupIdPtr )
100 {
101     Dio_PortLevelType result;
102
103         if(channelGroupIdPtr->port > no_of_ports)
104         {
105             //todo check channel id, if not valid Report_error(DIO_E_PARAM_INVALID_CHANNEL_ID)
106         }
107         else
108         {
109             result = ((*(DIO_u8pins[channelGroupIdPtr->port])) >> channelGroupIdPtr->offset ) & channelGroupIdPtr->masl
110         }
111
112         return result ;
113
114 }
115
116
117
118 void Dio_WriteChannelGroup(const Dio_ChannelGroupType *channelGroupIdPtr, Dio_PortLevelType level)
119 {
120     if(channelGroupIdPtr->port > no_of_ports)
121     {
122         //todo check channel id, if not valid Report_error(DIO_E_PARAM_INVALID_CHANNEL_ID)
123     }
124     else
125     {
126         *(DIO_u8ports[channelGroupIdPtr->port]) =  (~channelGroupIdPtr->mask) << channelGroupIdPtr->offset ; // for exa
127         *(DIO_u8ports[channelGroupIdPtr->port]) |= (level << channelGroupIdPtr->offset ) ;
128     }
129 }
130
131
```

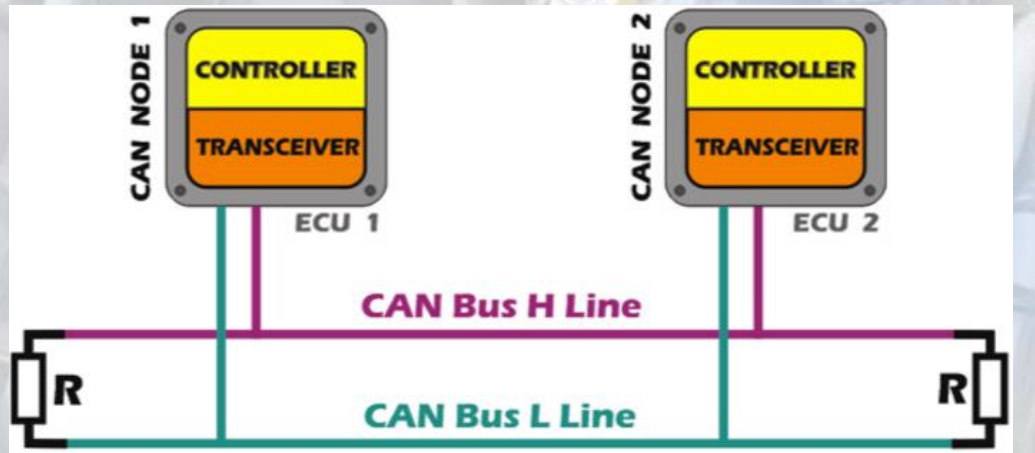
What is CAN ?

- Controller – Area – Network
- Developed in 1983 by Robert Bosch
  - To solve the networking issues in automotive
- Main Benefits
  - Economical
  - Reliable
  - Real Time response
  - Scalable
- Standards
  - CAN 2.0A (ISO11519)
  - Can 2.0B(ISO11898)



CAN-Leading Choice for Embedded Networking

- The main Reasons are
  - Economical
  - Reliability
    - Error Free Communication
    - Immune to EMI/EMS
  - Availability
    - Several 8/16/32 bit MCU available in the market
    - Standard development tools



Question

- Please give 3 reasons for the growing popularity of CAN in embedded applications
  - Reliability (works well in noisy environment)
  - Economical (Have low wiring costs)
  - Scalability
  - Availability

CAN Outlines

- It operates at transfer rates up to 1 Megabit/sec (1 Mbps) in CAN 2.0B. This speed provides sufficient data-communication bandwidth for many real-time control systems.
- The CAN protocol allows each CAN data frame to carry from zero to as many as eight bytes of user data per message, thus accommodating a wide span of signaling requirements. If necessary, more data can be transmitted per message using a higher-layer segmentation protocol.
- Each node on a CAN network can have several buffers or message mailboxes.

  On initialization, each mailbox is assigned an identifier that is either unique or is shared with certain other nodes. Also, each node is individually configured as a transmitter or receiver. This approach offers considerable flexibility in system design.

- messages are labeled by an identifier (ID) assigned one or more nodes on the network. All nodes receive the message and perform a filtering operation. That is, each node executes an acceptance test on the identifier to determine if the message — and thus its content — is relevant to that particular node. Only the node(s) for which the message is relevant will process it. All others ignore the message.

  The identifier has two more functions, as well. It contains data that specifies the priority of the message and it allows the hardware to arbitrate for the bus.
- Every node on the bus validates every message. Corrupted messages aren't validated, of course, and that situation triggers automatic re-transmissions.

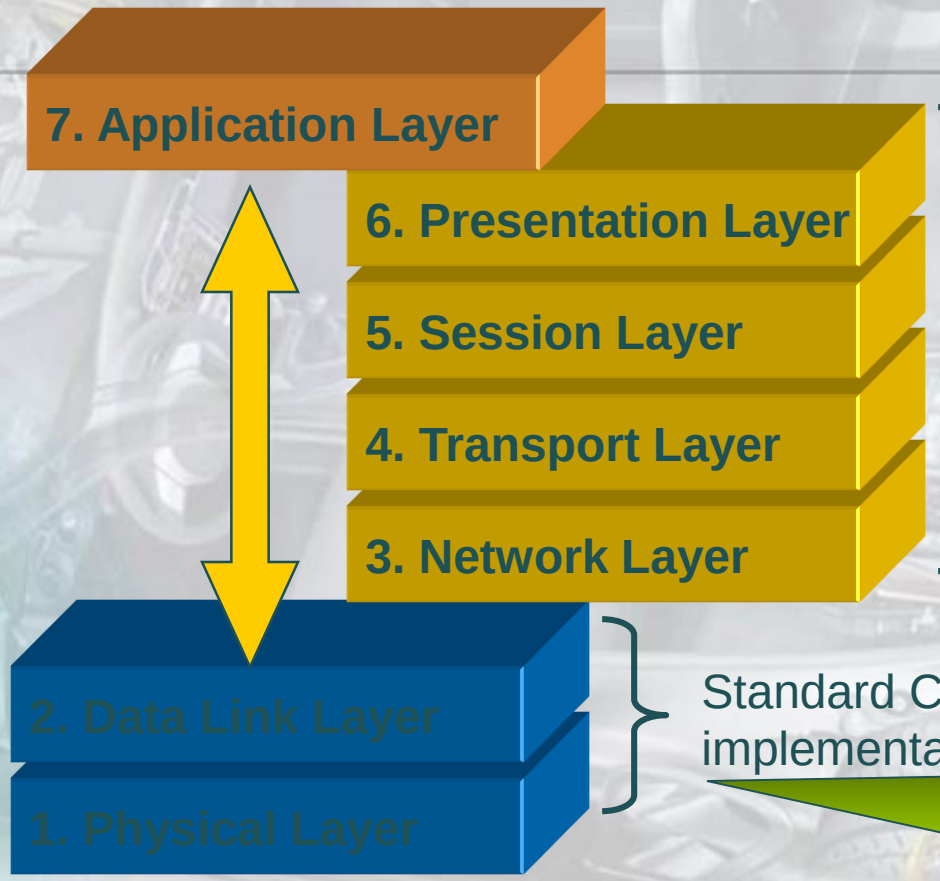
Features and Benefits of CAN

- Multiple Master Hierarchy
- 1 Mbps of Data transfer rate
- 0-8 Bytes of User Data
- Unique mail box Identifiers
- Acceptance Filtering by nodes
- Provides Error Detection
- Fault Confinement measures
- Auto re-transmit if corrupted

- Real Time Response
- Simplifies design requirements
- Flexibility in System Design
- Arbitration & Prioritization
- Ensures high Reliability
- Accurate communication link

CAN and the 7-layer model

ISA/OSI Reference Model



Partially implemented by higher-level CAN protocols (CANOpen)

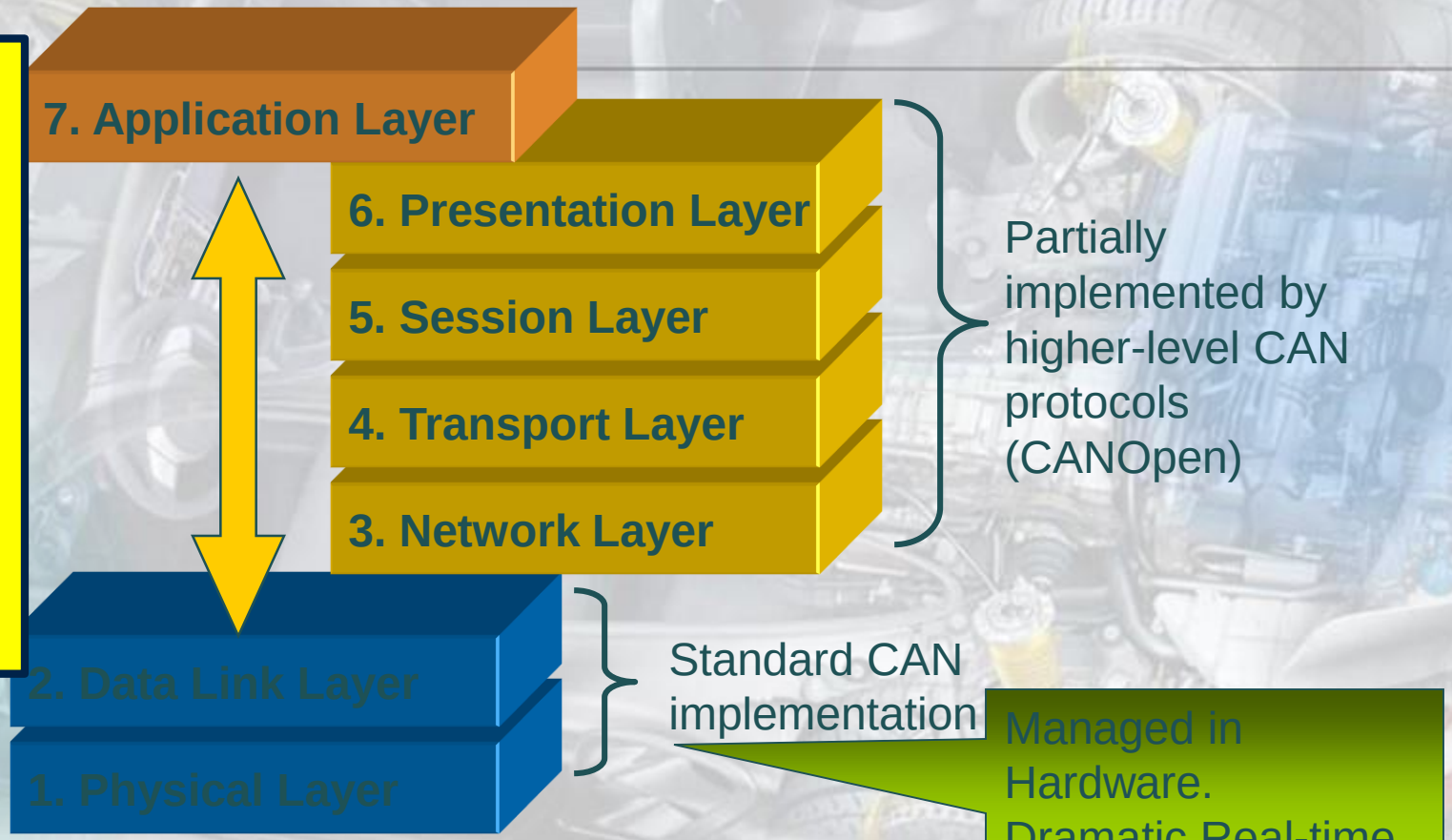
Standard CAN implementation

Managed in Hardware. Dramatic Real-time advantage to System Design

CAN and the 7-layer model

ISA/OSI Reference Model

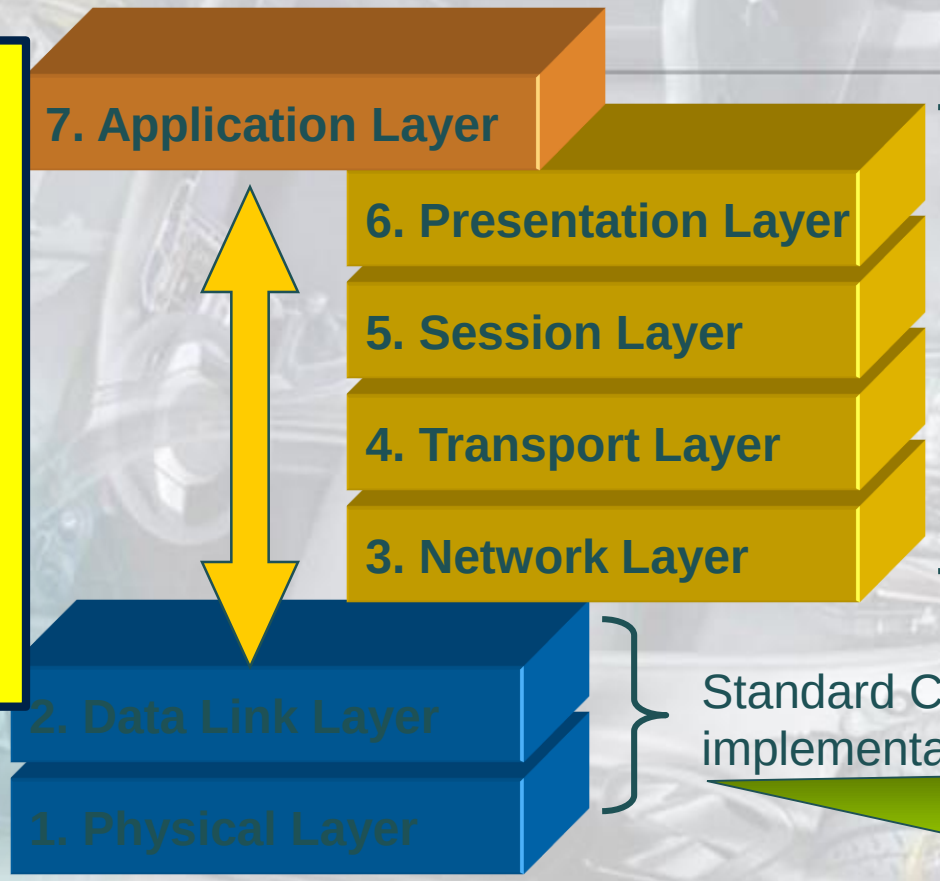
Having the Data Link Layer managed in hardware allows for more CPU cycles available for application management and better real-time control since time does not need to be allotted for simple message monitoring.



CAN and the 7-layer model

ISA/OSI Reference Model

The standard CAN implementation bypasses the connection between the Data Link layer and the Application layer. The layers above the Data Link Layer are implemented in software which as per definition are called the Higher Layer Protocol

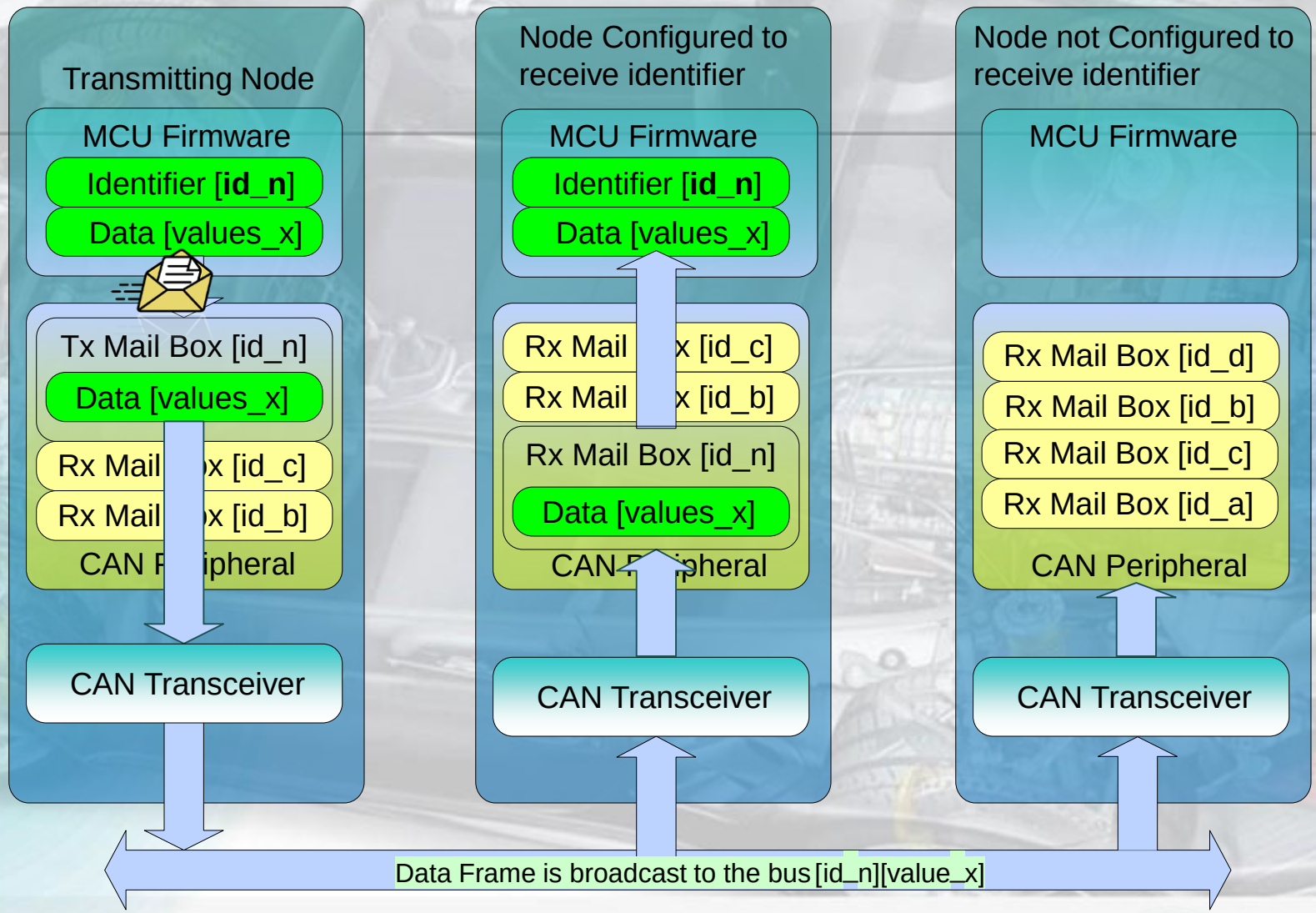


Partially implemented by higher-level CAN protocols (CANOpen)

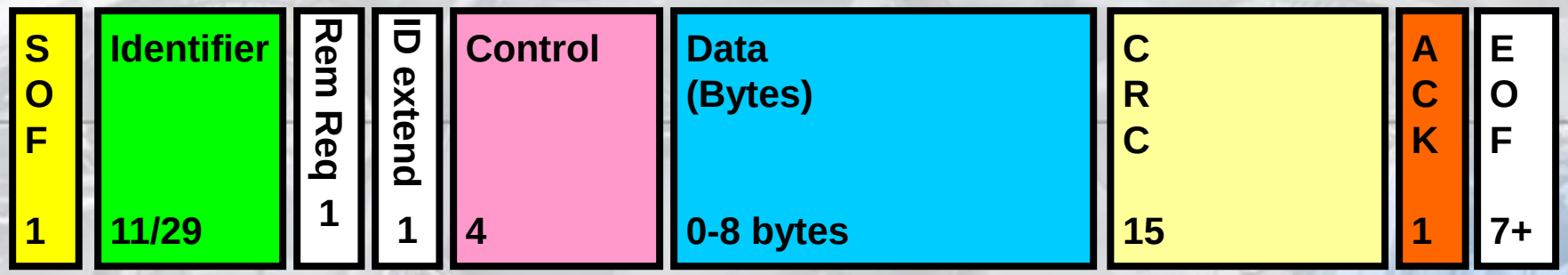
Standard CAN implementation

Managed in Hardware. Dramatic Real-time advantage to System Design

Data Flow in CAN



Data Frame

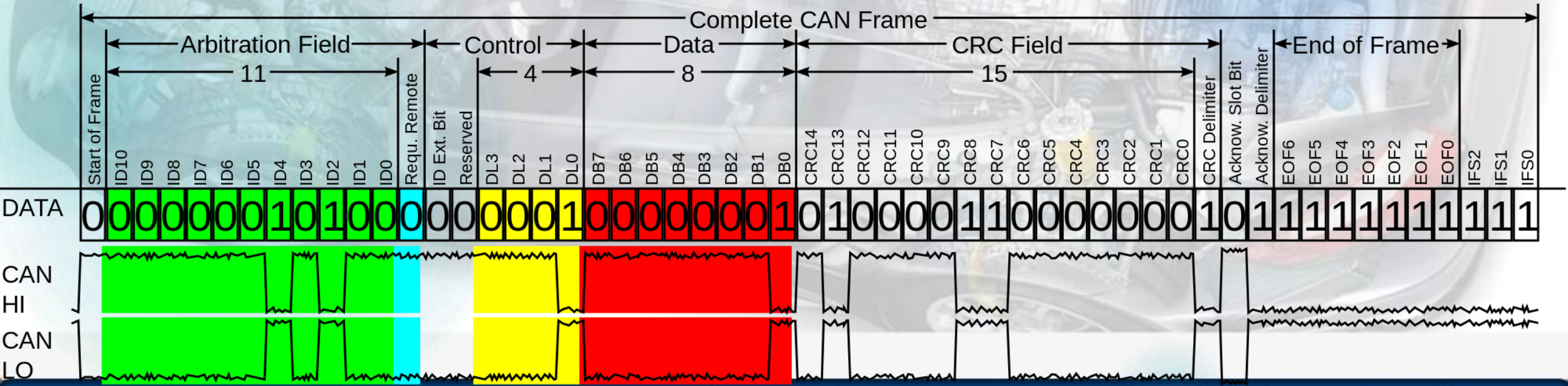


- Start of Frame – 1-bit
- Arbitration Field – 11-bits/29-bits
- Control Field – 6 bits (2 reserved, 4 representing number of Data Field bytes)
- Data Field – 0 to 8 BYTES
- CRC – 15-bits
- ACK Field – 1-bit/variable
- End of Frame – 7-bits (recessive)

Start Of Frame [SOF]

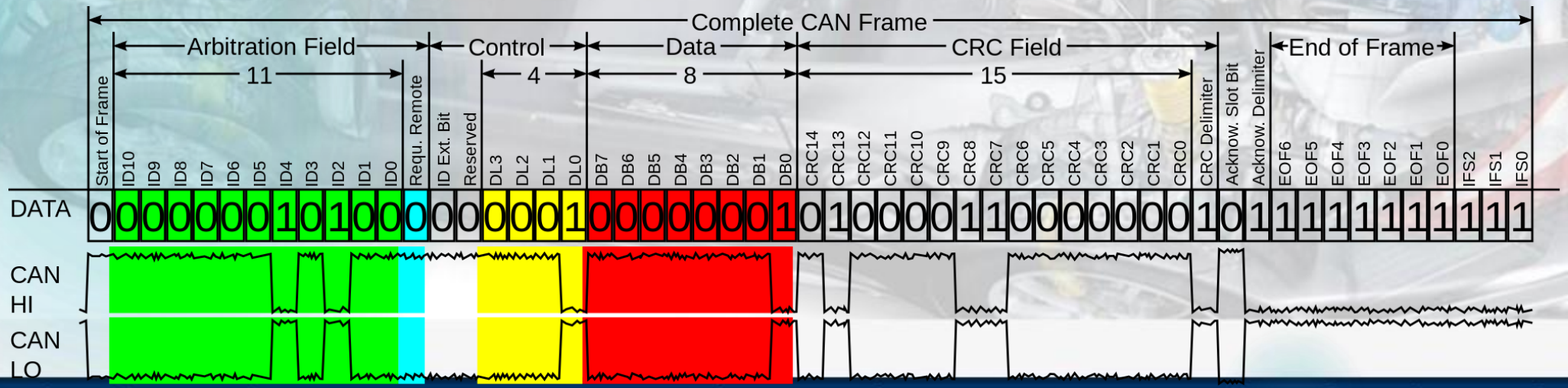
- SOF: CAN has a multi-master capability, meaning any node on the bus can initiate communication to any node configured to receive. This is done with a Start of Frame. A single dominant bit while the bus is idle indicates a transmitting node is starting a frame.

All nodes on the bus will synchronize their bit timing to the leading edge of SOF.



Arbitration Field: ID, or identifier of CAN 2.0B

- may be 11 or 29 bits long This is determined by FW at initialization by setting the ID-extension bit (and then configuring each can mailbox with a long ID instead of a standard). Extended ID is less common **but you need extended ID in case you have more than 2032 IDs to differentiate.**
- CAN has a multi-master capability meaning any node on the bus can initiate communication to any other node. This is where the Arbitration Field comes in...

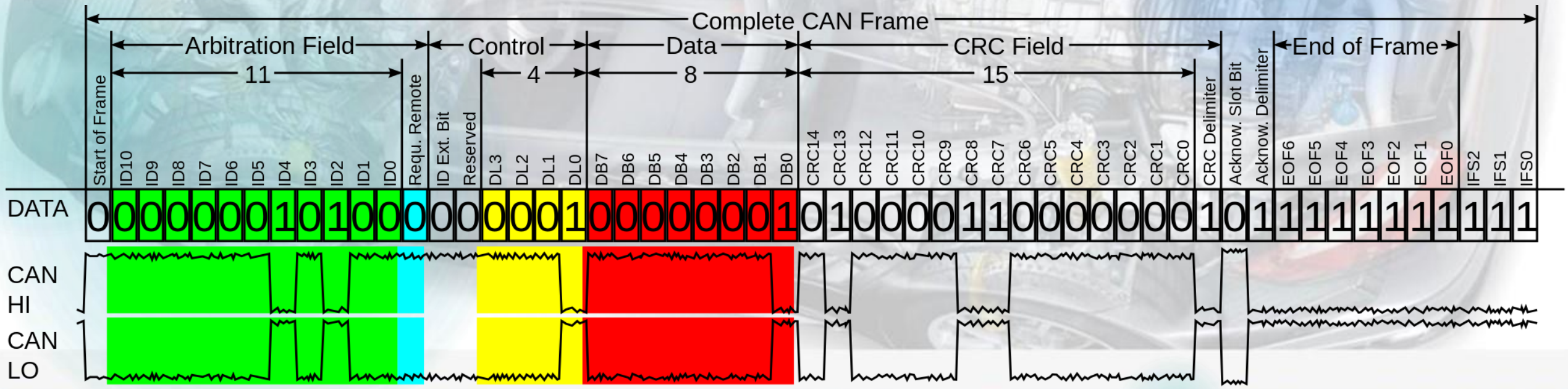


Arbitration Field: ID, or identifier of CAN 2.0B

- the extended ID is split into two pieces; a high order 11 bit field, and a low order 18 bit field. Because the ID extend bit is recessive in the extended format, that the standard ID has a higher priority than the extended ID for the same leading 11 bit identifier.
- To allow multiple devices to initiate communication, the arbitration protocol determines which device will receive priority and access to the bus. This is called Carrier Sense Multiple Access/Collision Resolution (CSMA/CR).
- When two nodes negotiate for the bus at the same time, a dominant bit state will override a recessive bit state. When a node transmits a recessive state on the bus, but detects a dominant state on its receiver, it knows that higher priority message is being transmitted and immediately ceases transmission. The losing node will renegotiate for the bus as each new opportunity comes until the message is transmitted.

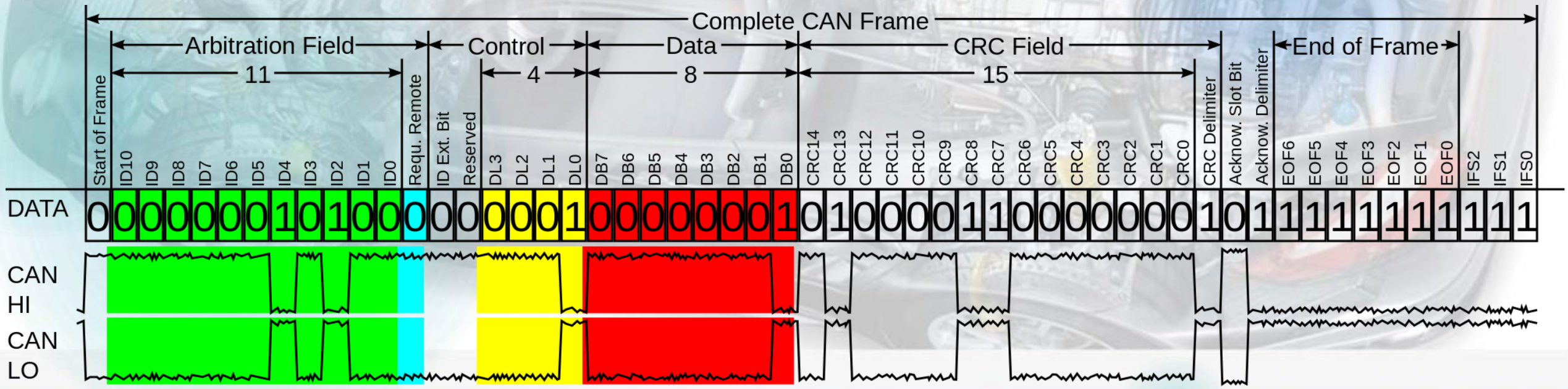
Control Field

- The Control Field consists of six bits
  - - 2 are reserved
  - - 4 are the Data Length Code which indicates number of data bytes in the data field [0-8 bytes being valid sizes]
  - DLC codes not shown in the figure are reserved



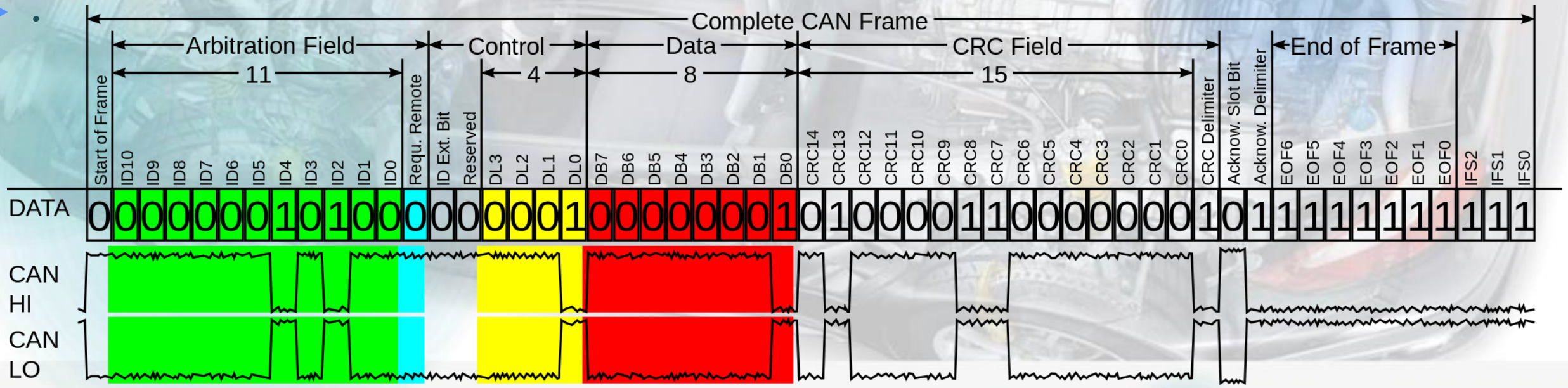
Data

- This field carries the actual payload of the can-bus communication. It may be 0 to 8 bytes long, as defined by the Data Length Code in the Control Field. The most significant bit is transmitted first within each byte.



CRC

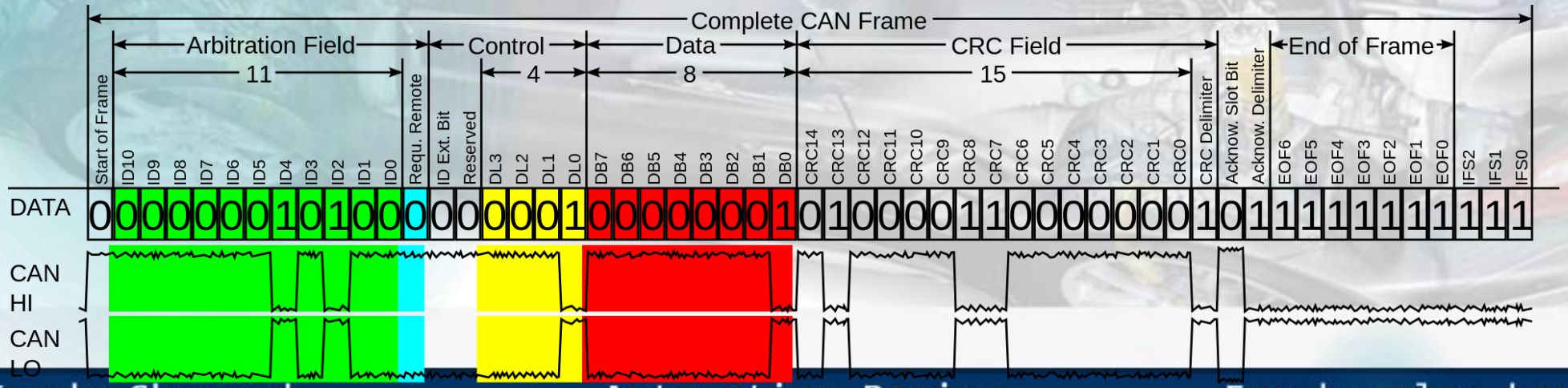
- The CRC Field is sent by the transmitter and verified by all receivers. Each receiver generates a CRC on the observed data frame and compares it with the transmitted CRC check value. If it is they match, a dominant bit is put into the ACK slot. If the result is a mismatch, nothing is transmitted by that node; instead, a 'no' vote is sent AFTER the upcoming ACK delimiter, that is at End of Frame.
- .



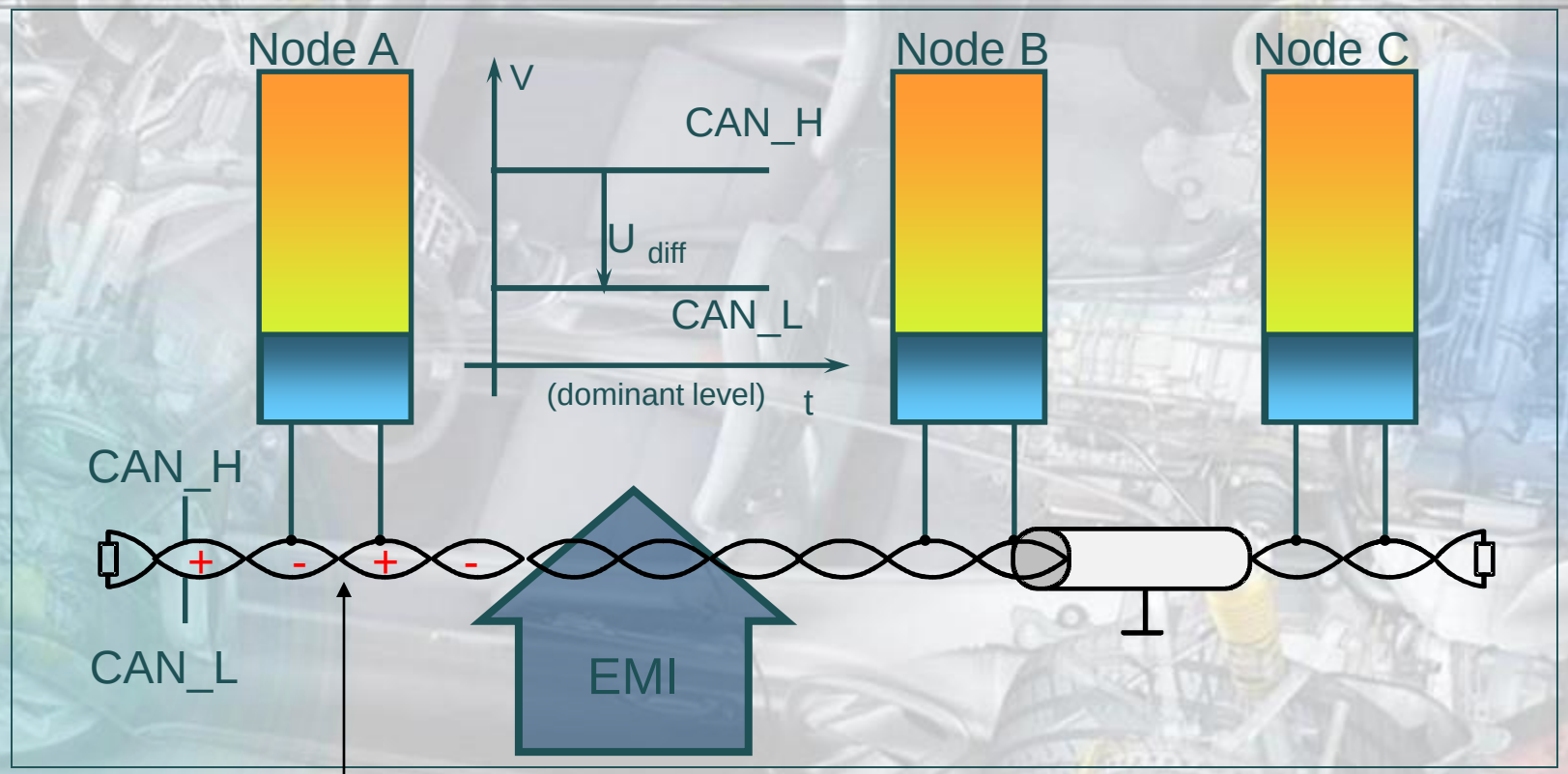
End Of Frame

- The ACK FIELD is two bits long and contains the ACK slot and the ACK delimiter. In the ACK slot the transmitting station sends two recessive bits. Every receiver which has received a valid message correctly, reports this to the transmitting node with a 'dominant' bit during the ACK slot. Any node that disagrees, votes no after the delimiter by sending an error flag.

- This is the last of the fields of the data frame. The End of Frame field provides the necessary portion of idle between messages, for example for the transmitter to detect if a node decided to send an error frame, which is allowed anytime.
- The End of Frame field consists of seven recessive bits.

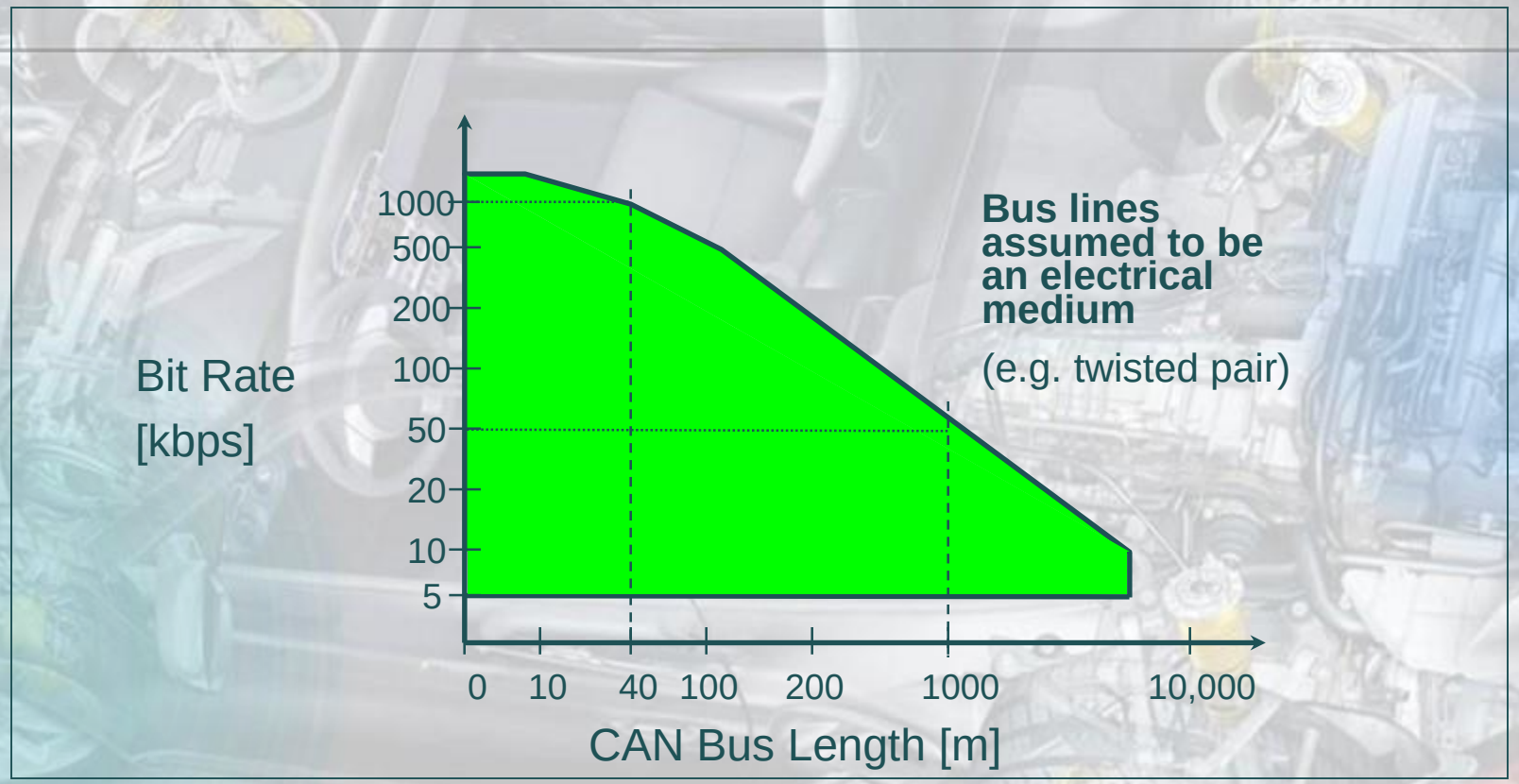


CAN and EMI



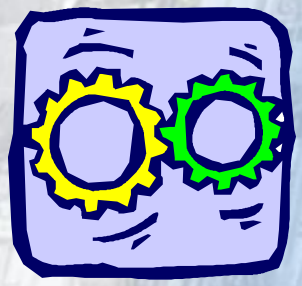
CAN-Bus
(Differential Serial Bus)

CAN Baud Rate vs. Bus Length

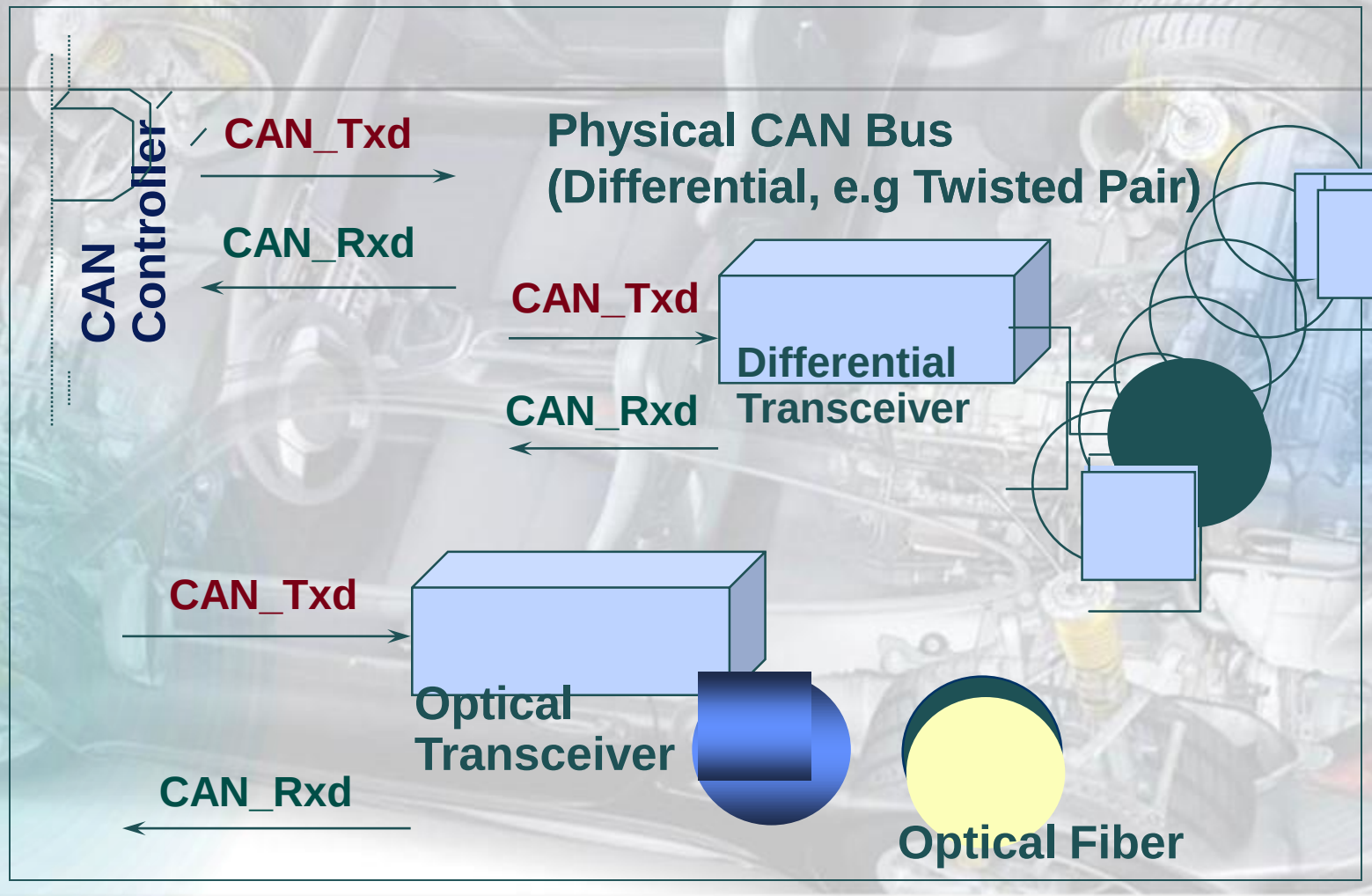


Error Detection in CAN

- Error statistics depend up on the entire environment
  - Total number of nodes
  - Physical Layout
  - EMI Disturbance
- CAN application example running at
  - 2000 hours/year, 500 Kbps, 25% Bus load
  - Results in one undetected error in 1000 years



Physical Layer



What is LIN ?

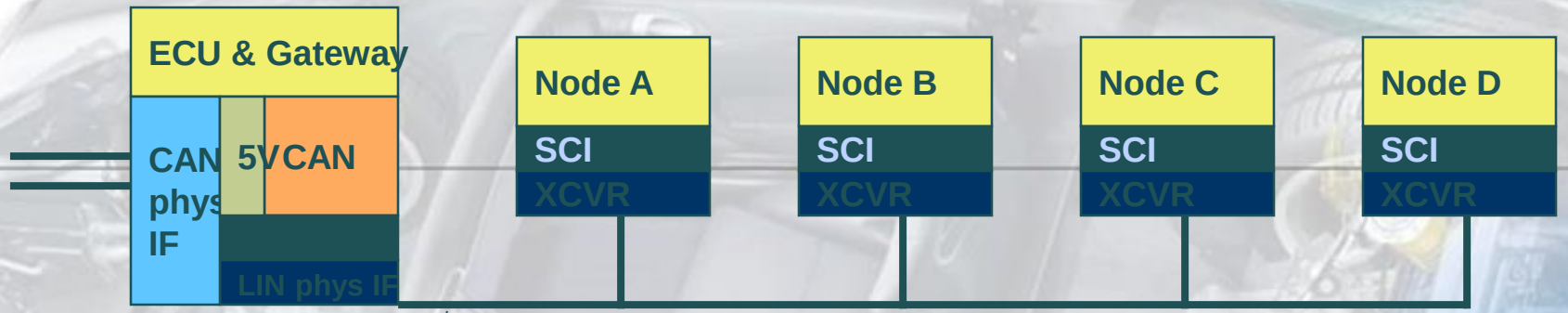
- Local Interconnect Network
  - A slower & low cost alternative to CAN
- Developed by LIN Consortium in 2002
- Developed as a sub-network of CAN to reduce the Bus Load
- Applications
  - Automotive, White Goods, Medical – for sensors and actuators

Features & Benefits of LIN

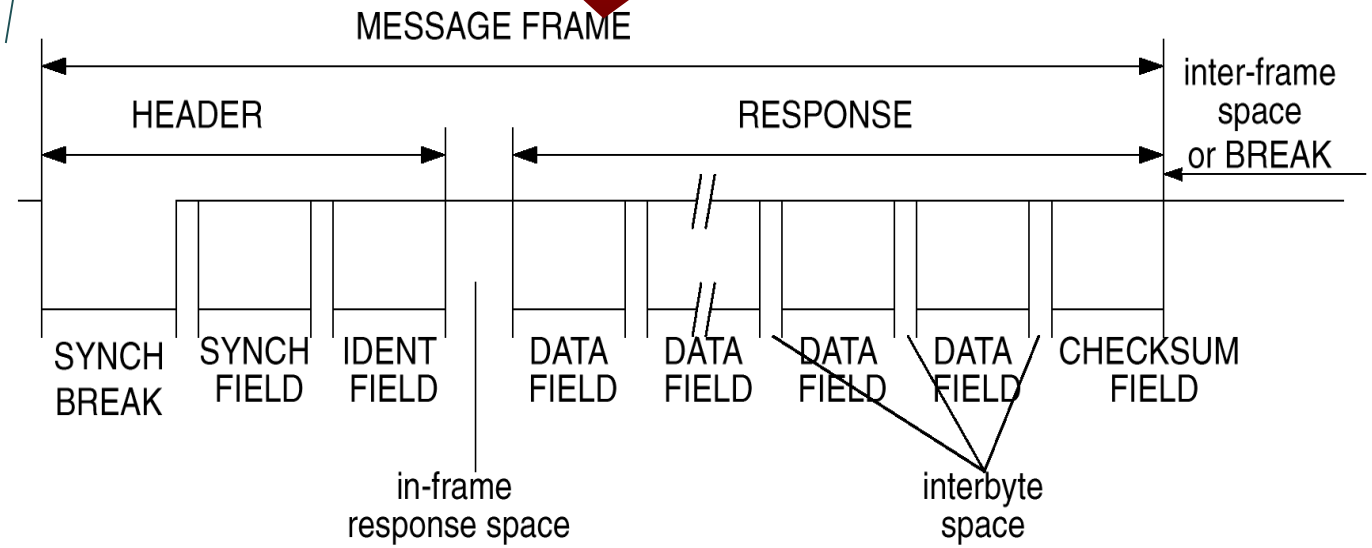
- Complementary to CAN
- Single Wire Implementation
- Speed up to 20Kbps
- Single Master/Multiple Slave
- Based on common UART/SCI
- Self Synchronization
- Guaranteed latency times

- Extends CAN to sub-nets
- Reduce harness costs
- Improves EMI response
- No arbitration necessary
- Reduces risk of availability
- No external crystal
- Deterministic & Predictable

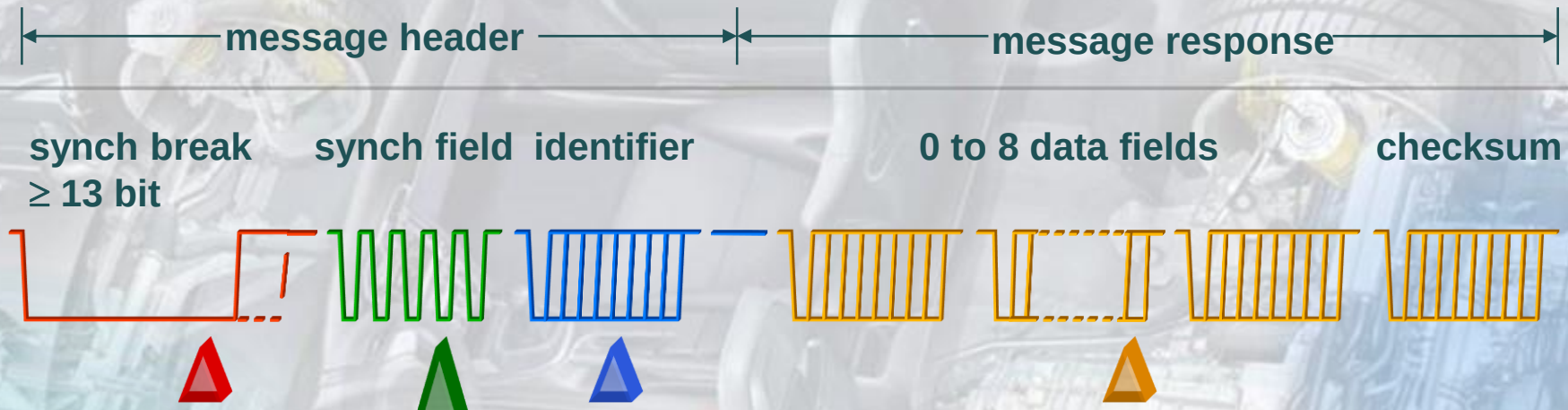
Typical LIN Network



Simplex
12V Operation



LIN Message Frame



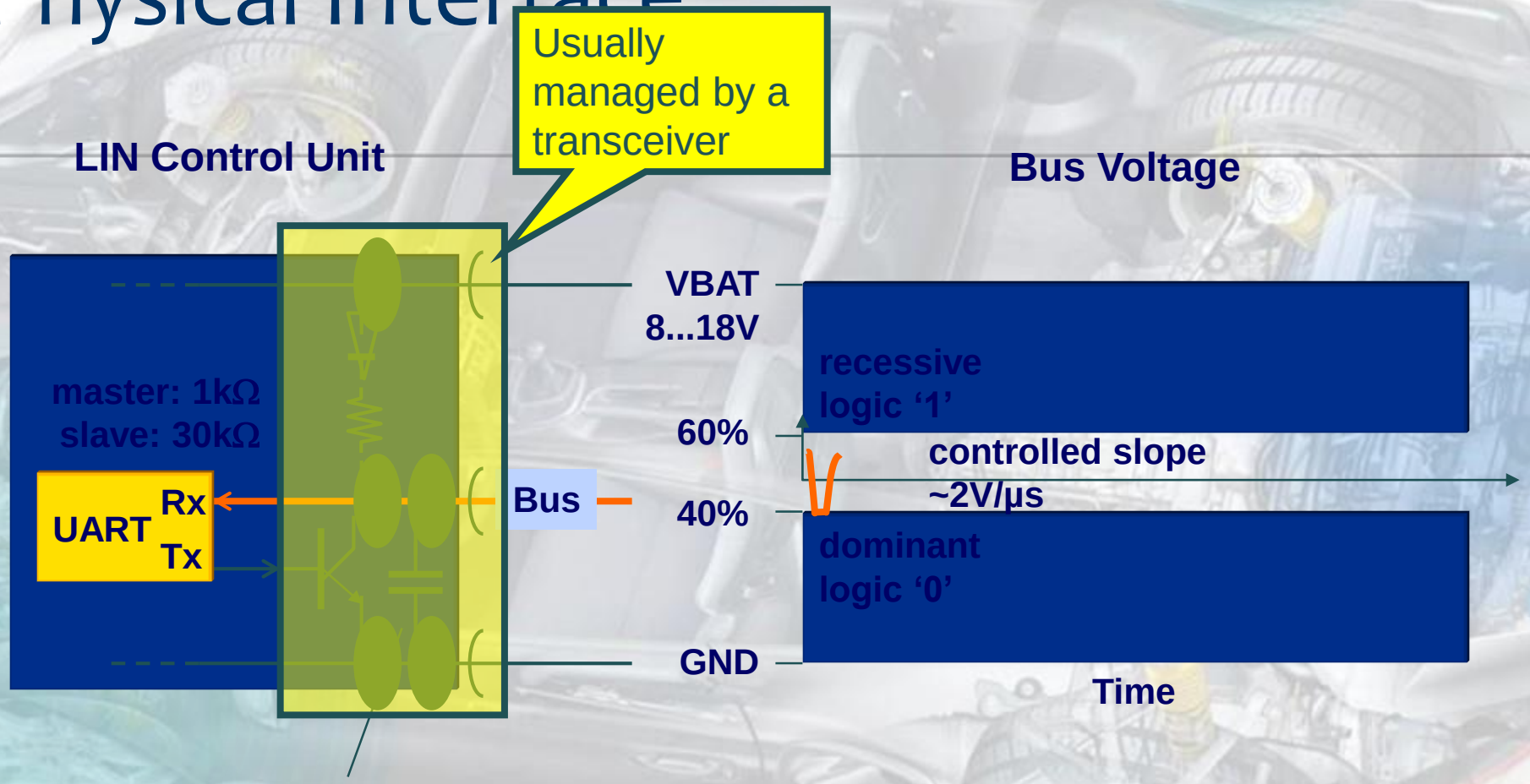
Synchronization Frame

Synchronization Field

Identifier Byte

Message

LIN Physical Interface



Question

- What are the reasons when LIN is preferred over CAN?
  - To save the bandwidth of another main bus
  - Size of Network is 16 nodes or less
  - When lower speed is acceptable
  - Economical
  - Single Master with multiple slaves

LIN versus CAN

	LIN *versus* CAN	
Access Control	Single Master	Multiple Master
Max Bus Speed	20 Kbps	1 Mbps
Typical # nodes	2 to 16	4 to 20
Message Routing	6-bit Identifier	11/29-bit Identifier
Data byte/frame	2,4,8 bytes	0-8 bytes
Error detection	8-bit checksum	16-bit CRC
Physical Layer	Single-wire	Twisted-pair


Thank
You

References

- http://www.autosar.org/about/technical-overview/ecu-software-architecture/autosar-basic-software/
- http://www.autosar.org/standards/classic-platform/
- https://automotivetechis.files.wordpress.com/2012/05/communicationstack_gosda.pdf
- https://automotivetechis.files.wordpress.com/2012/05/autosar_ppt.pdf
- https://automotivetechis.wordpress.com/autosar-concepts/
- https://automotivetechis.files.wordpress.com/2012/05/autosar_exp_layeredsoftwarearchitecture.pdf
- http://www.slideshare.net/FarzadSadeghi1/autosar-software-component
- https://www.renesas.com/en-us/solutions/automotive/technology/autosar/autosar-mcal.html
- https://github.com/parai/OpenSAR/blob/master/include/Std_Types.h