



## ECU软件的AUTOSAR分层架构

浙江大学ESE工程中心



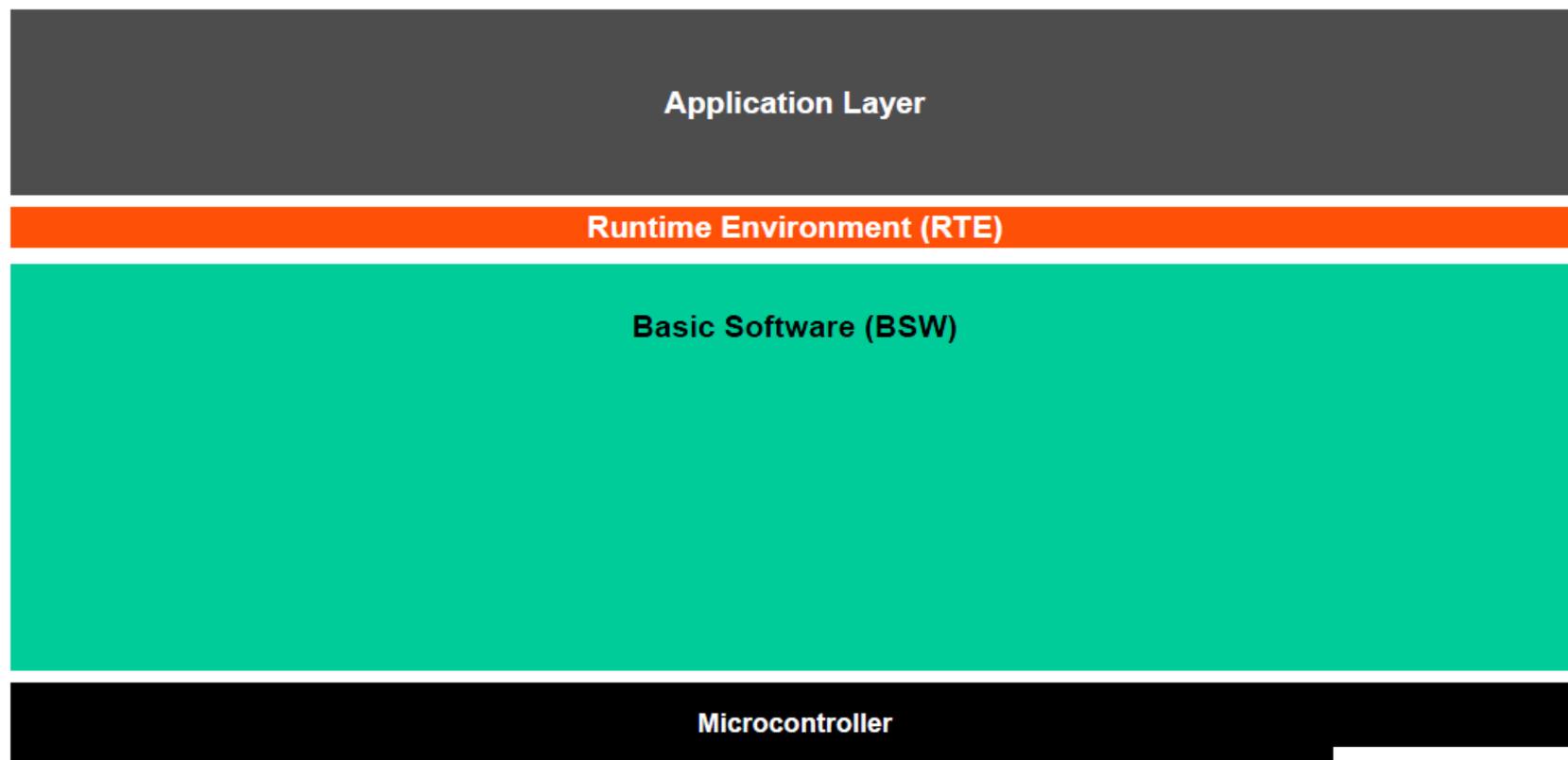
# Outline

- 分层概述
- 应用层
- VFB与RTE层
- 基础软件（BSW）
- 示例



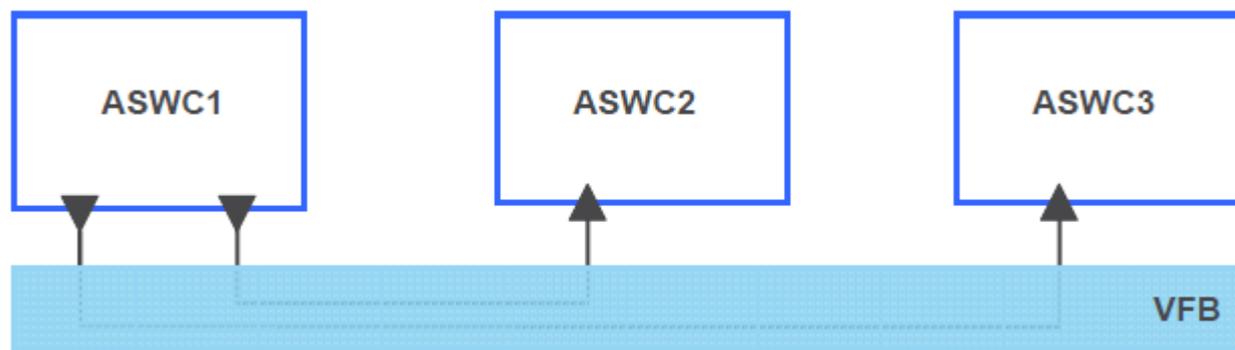
# 概述

- 将运行在Microcontroller之上的ECU软件分为Application、RTE、BSW三层



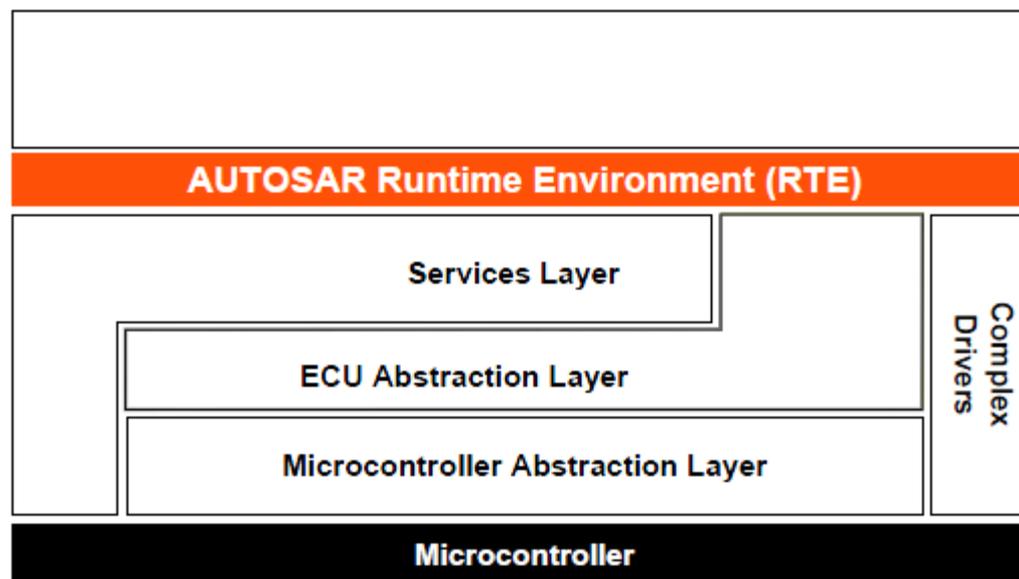
# 应用层

- 应用层将软件都划分为一个Atomic Software component (ASWC)，包括硬件无关的Application Software Component、Sensor Software Component、Actuator Software Component、等。



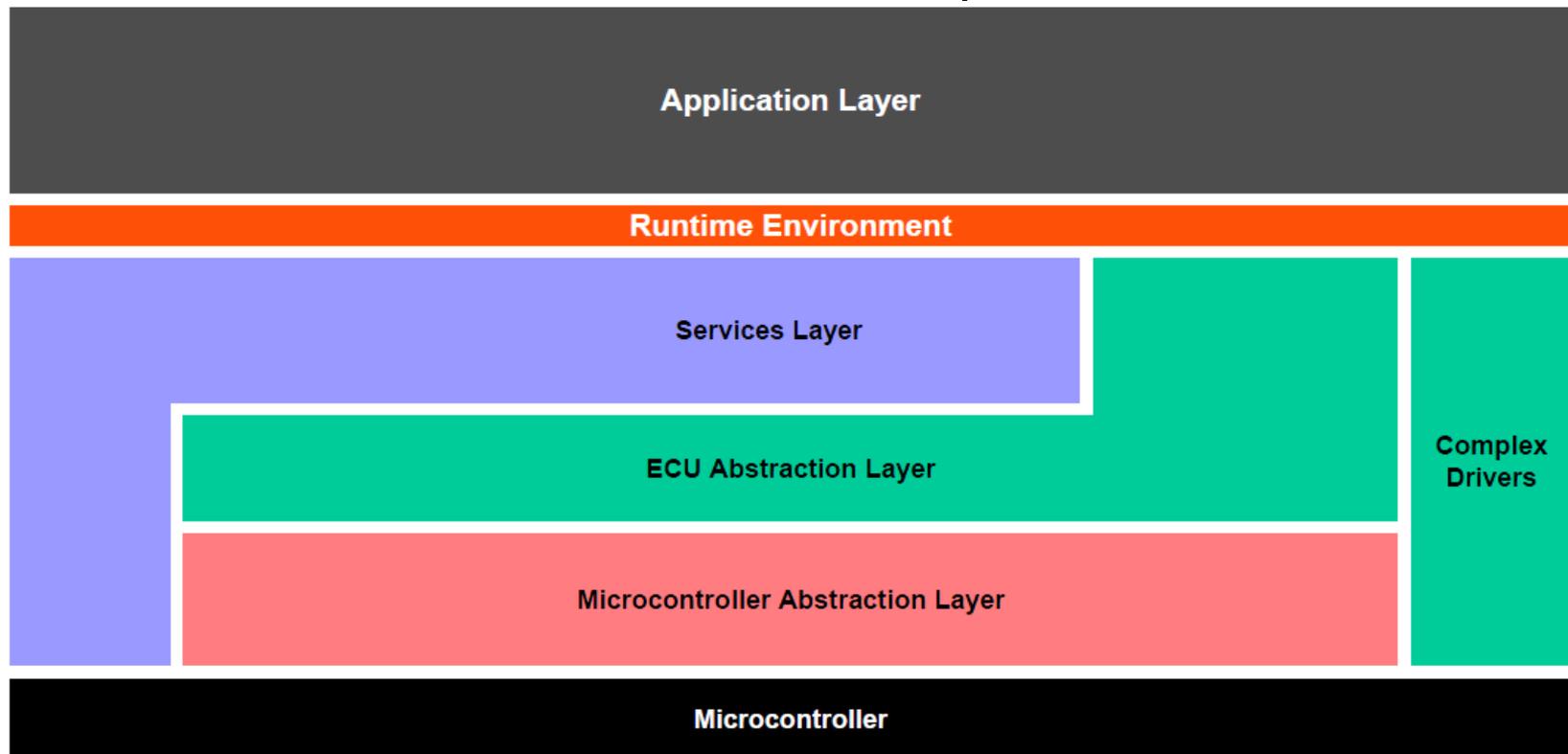
# RTE

- RTE提供基础的通信服务，支持Software Component之间和Software Component到BSW的通信（包括ECU内部的程序调用、ECU外部的总线通信等情况）。
- RTE使应用层的软件架构完全脱离于具体的单个ECU和BSW。



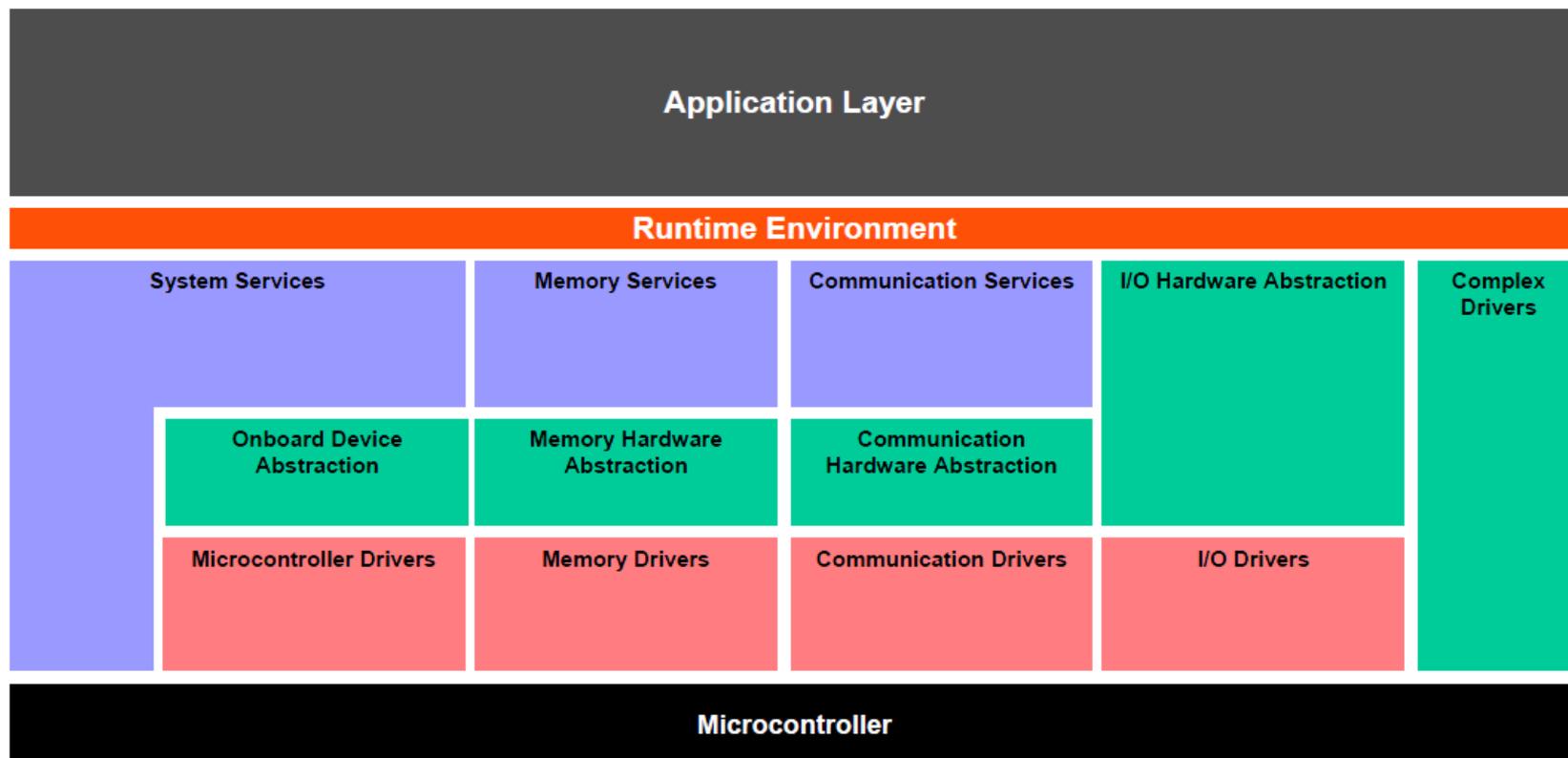
# BSW层

- 将基础软件层(BSW)分为Service、ECU Abstraction、Microcontroller Abstraction以及Complex Drivers。



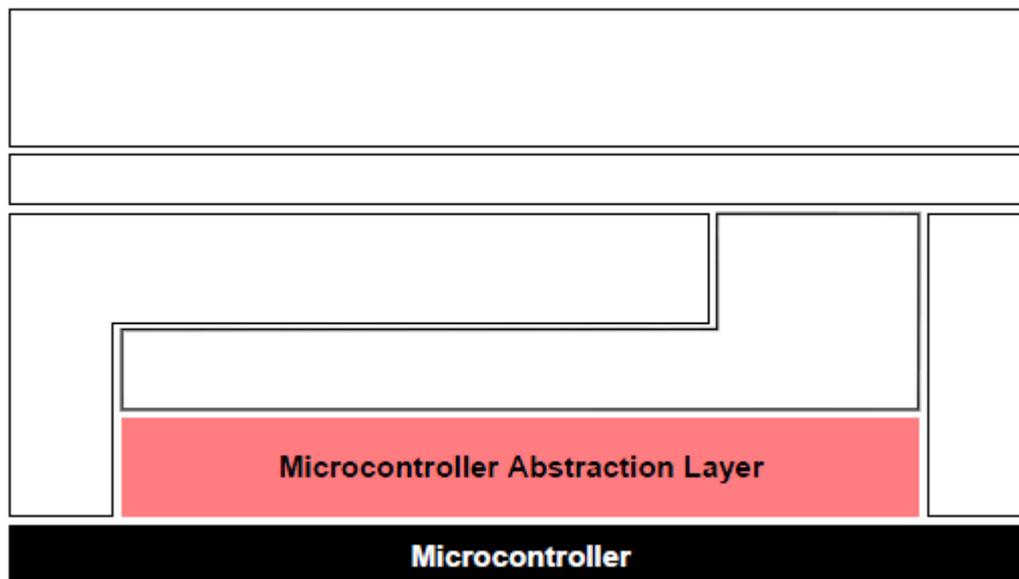
# BSW层的功能模块

- 每层的BSW中，都保护不同的功能模块。比如Service层包括系统服务、内存服务、通信服务。



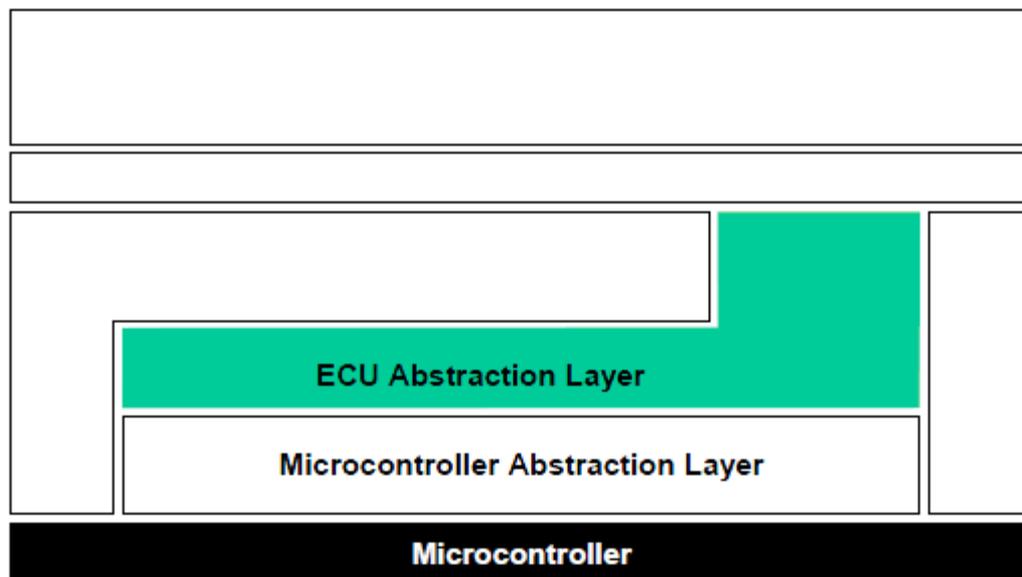
# BSW-微控制器抽象层

- 微控制器抽象层(Microcontroller Abstraction Layer) 是在BSW的最底层，它包含了访问微控制器的驱动。
- 微控制器抽象层使上层软件与微控制器相分离，以便应用的移植。



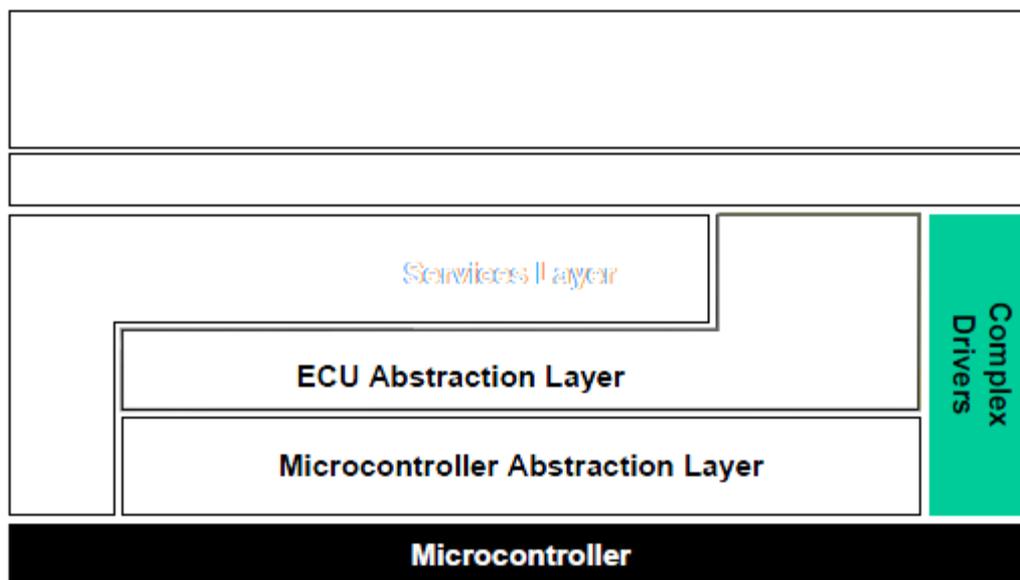
# BSW-ECU抽象层

- ECU抽象层封转了微控制器层以及外围设备的驱动。
- 将微控制器内外设的访问进行了统一，使上层软件应用与ECU硬件相剥离。



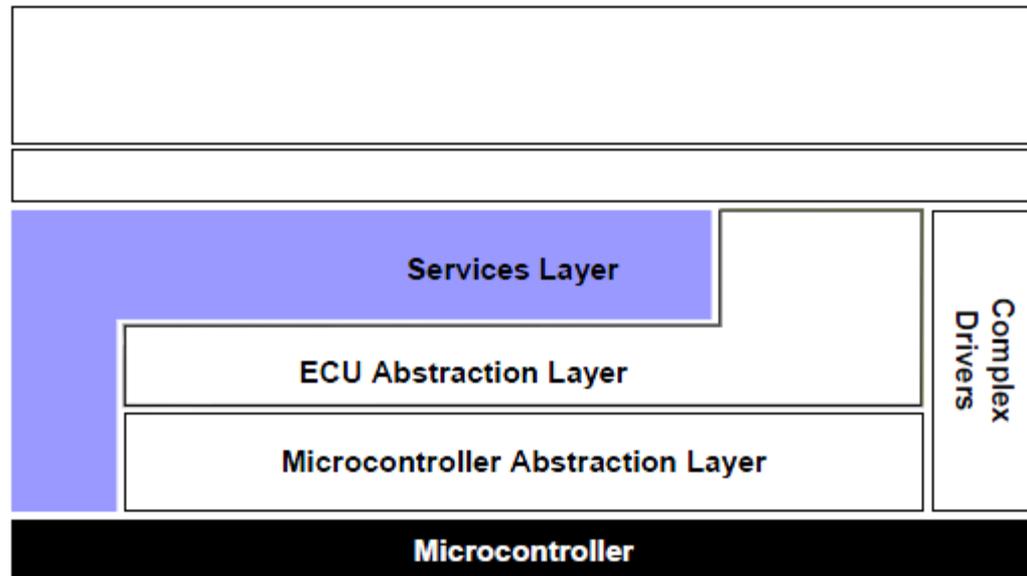
# BSW-复杂驱动

- 为了满足实时性等要求，可以利用复杂驱动（Complex Drivers），让应用层通过RTE直接访问硬件。
- 也可以利用复杂驱动封转已有的非分层的软件，以实现向AUTOSAR软件架构逐步实施。



# Service层

- 服务层（Service Layer）位于BSW的最上面，将各种基础软件功能以服务的形式封转起来，供应用层调用。
- 服务层包括了RTOS、通信与网络管理、内存管理、诊断服务、状态管理、程序监控等服务。



# Service的类型介绍

- BSW包括以下服务类型：
  - Input/output(I/O)服务: 将执行器、传感器以及外设的访问标准化
  - 内存服务: 将微控制器内外内存的访问进行统一封装
  - 通信服务: 将整车网络系统、ECU网络系统、软件组件内的访问进行统一封装
  - 系统服务: 包括RTOS、定时器、错误处理、看门狗、状态管理等服务

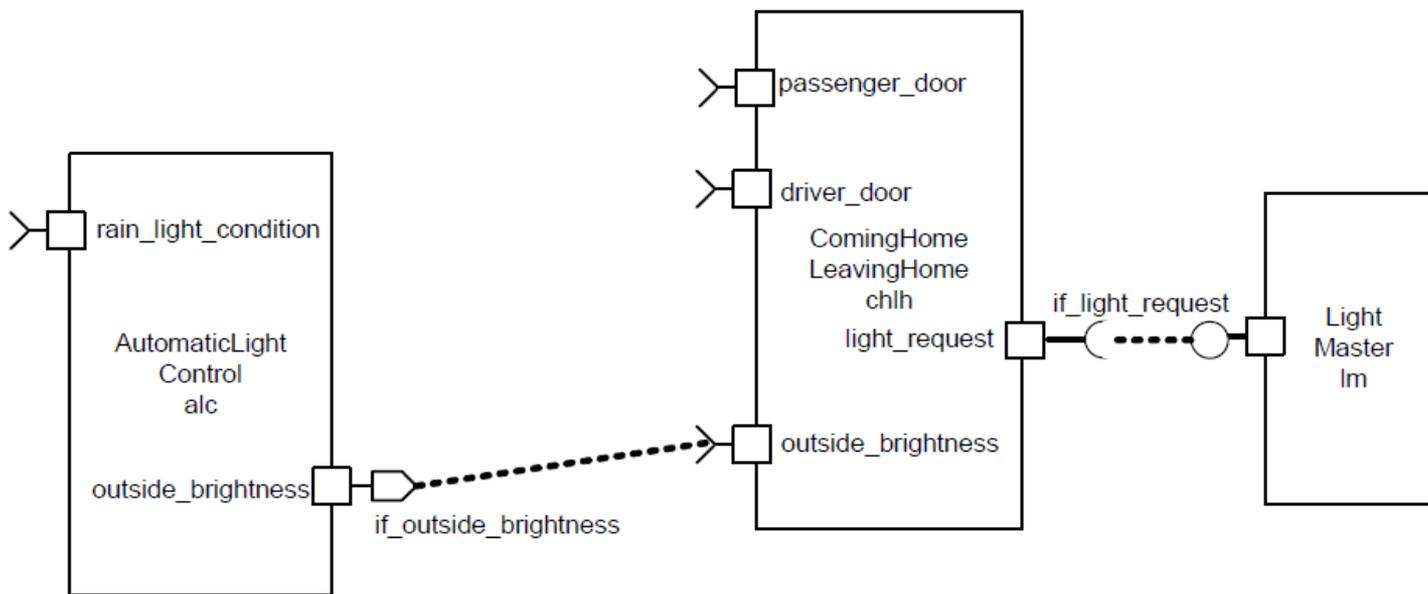
# Outline

- 分层概述
- 应用层
- VFB与RTE层
- 基础软件（BSW）
- 示例



# AUTOSAR Software Component

- 应用层由各种AUTOSAR Software Component (SW-C) 组成
- 每个AUTOSAR SW-C都封装了各种应用的功能集，可大可小
- 每个AUTOSAR SW-C只能运行在一个ECU中，也可称为Atomic SW-C

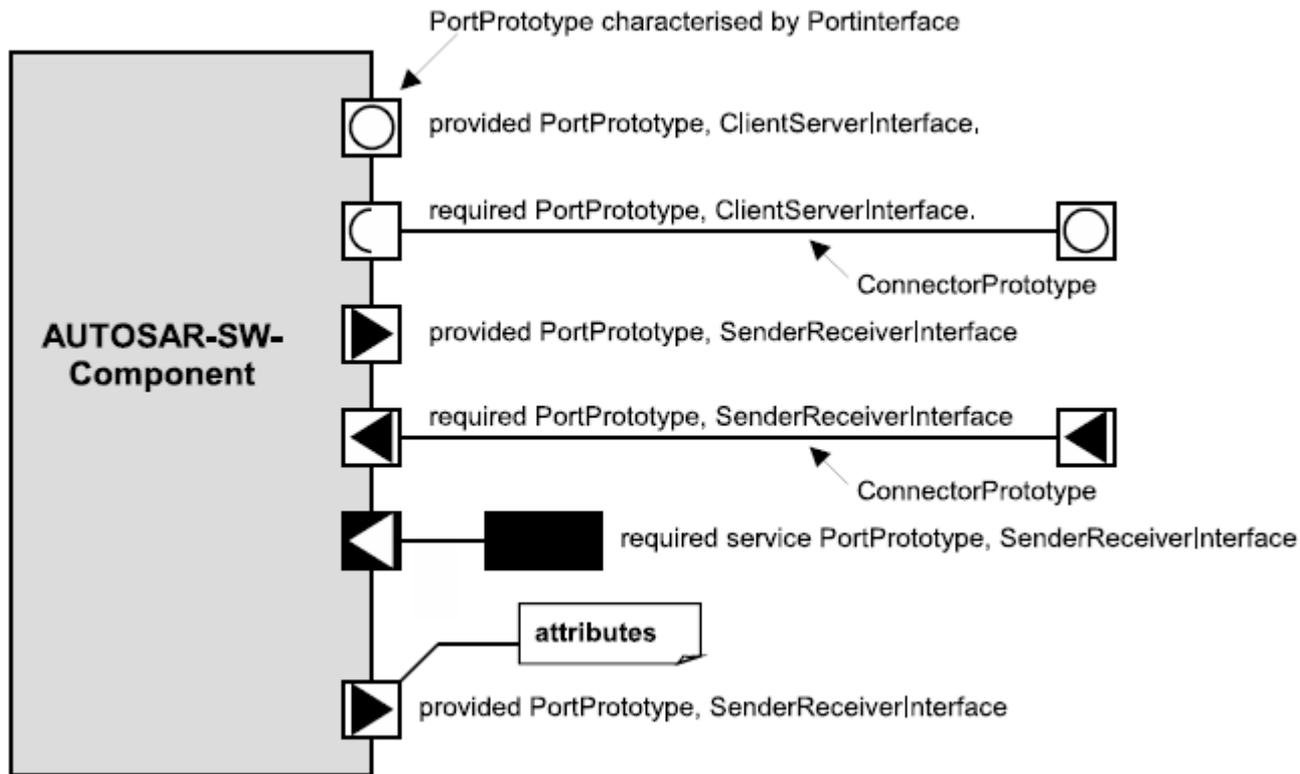


# SW-C的实现

- 可以通过算法建模、手写代码等多种方式实现SW-C。
- 在AUTOSAR架构体系中，SW-C的实现：
  - 与MCU类型无关
  - 与ECU类型无关
  - 与相互关联的SW-C的具体位置无关
  - 与具体SW-C的实例个数无关
- Software Component Template规定了SW-C的描述规范

# Port和Interface

- Port: 表示输入 (RPort) 或输出 (PPort)
- Interface: 具体输入输出的方式、数据类型等



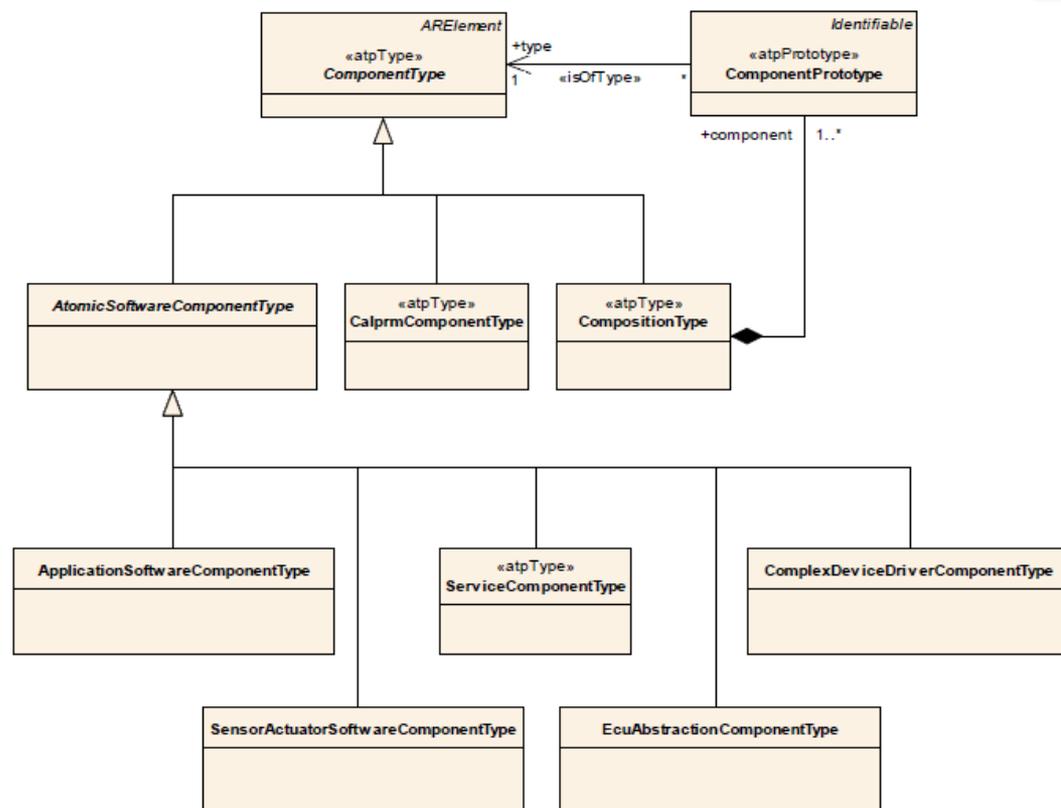
# SW-C的类型

## ■ 原子软件组件（ASWC）

- 应用软件组件
- 输入输出软件组件
- 服务组件
- ECU抽象组件
- 复杂驱动组件标定程序组件

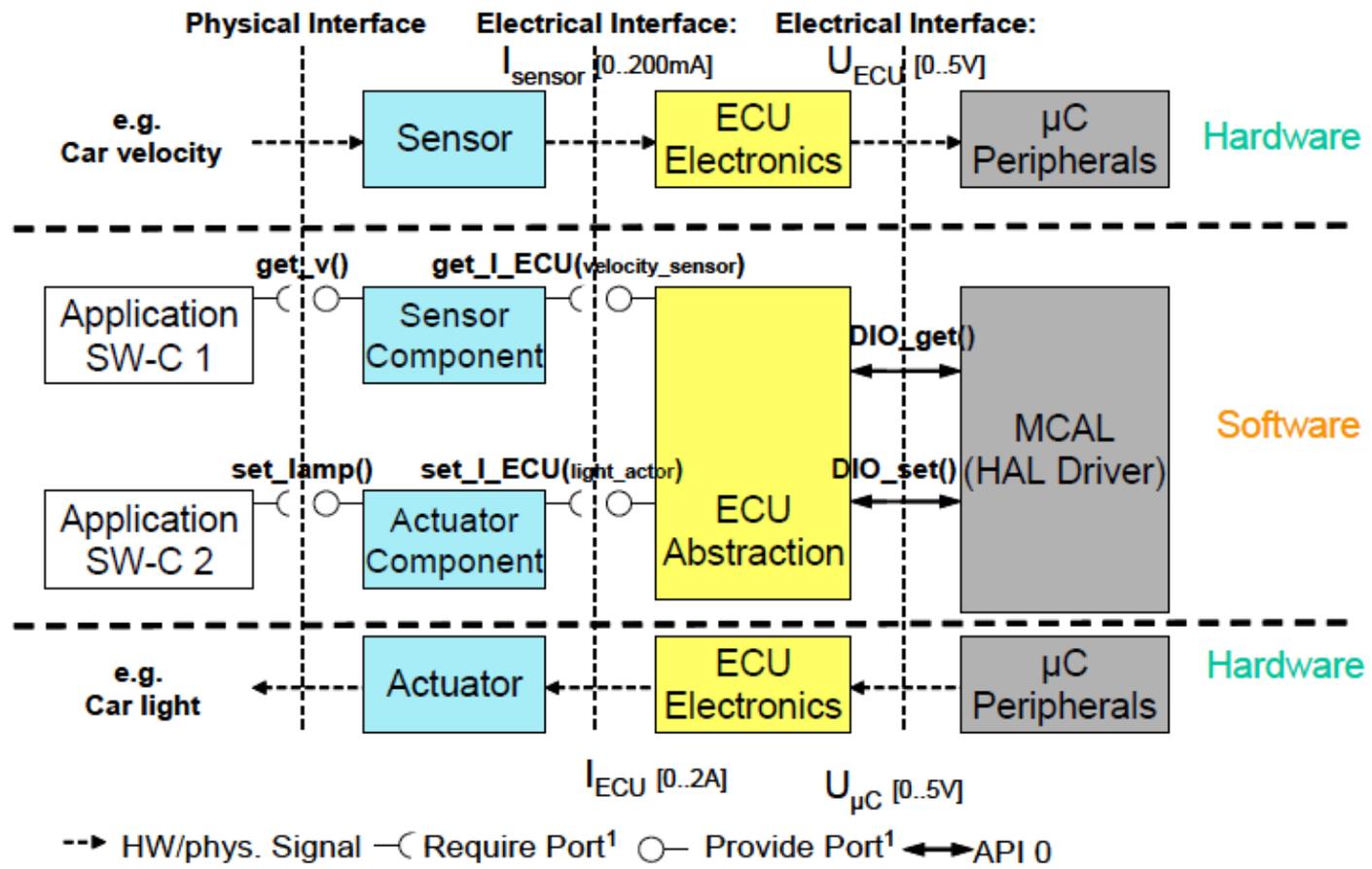
## ■ 标定程序组件

## ■ 组合Composition



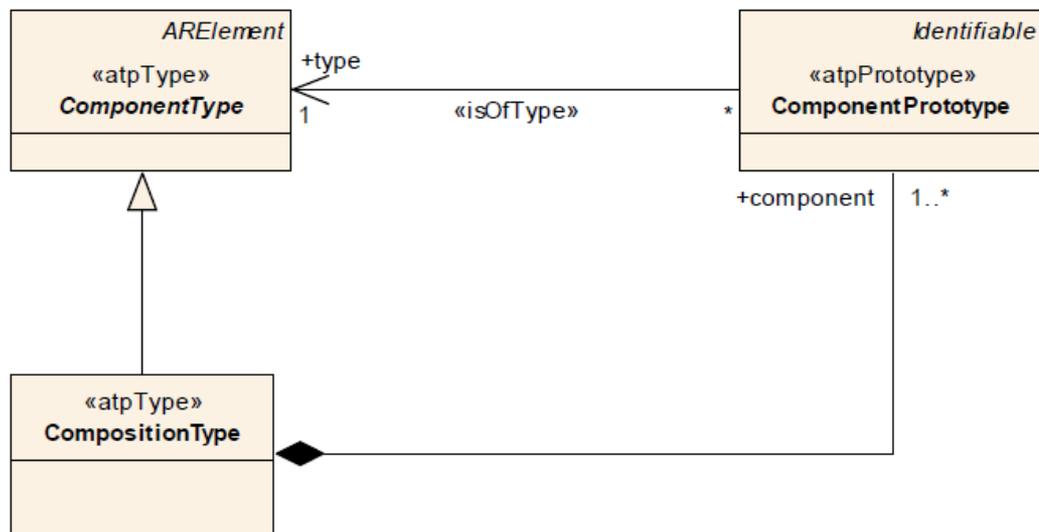
# Sensor/Actuator Software Components

- 所有I/O的输入输出都通过Sensor/Actuator SW-C



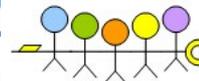
# Composition

- Composition是多个ASWC的实例集合，也当做是SW-C。
- Composition的Port是内部某个ASWC的Port代理，通过DelegationConnector来表示。
- Composition内ASWC之间的输入输出是通过AssemblyConnector来表示。



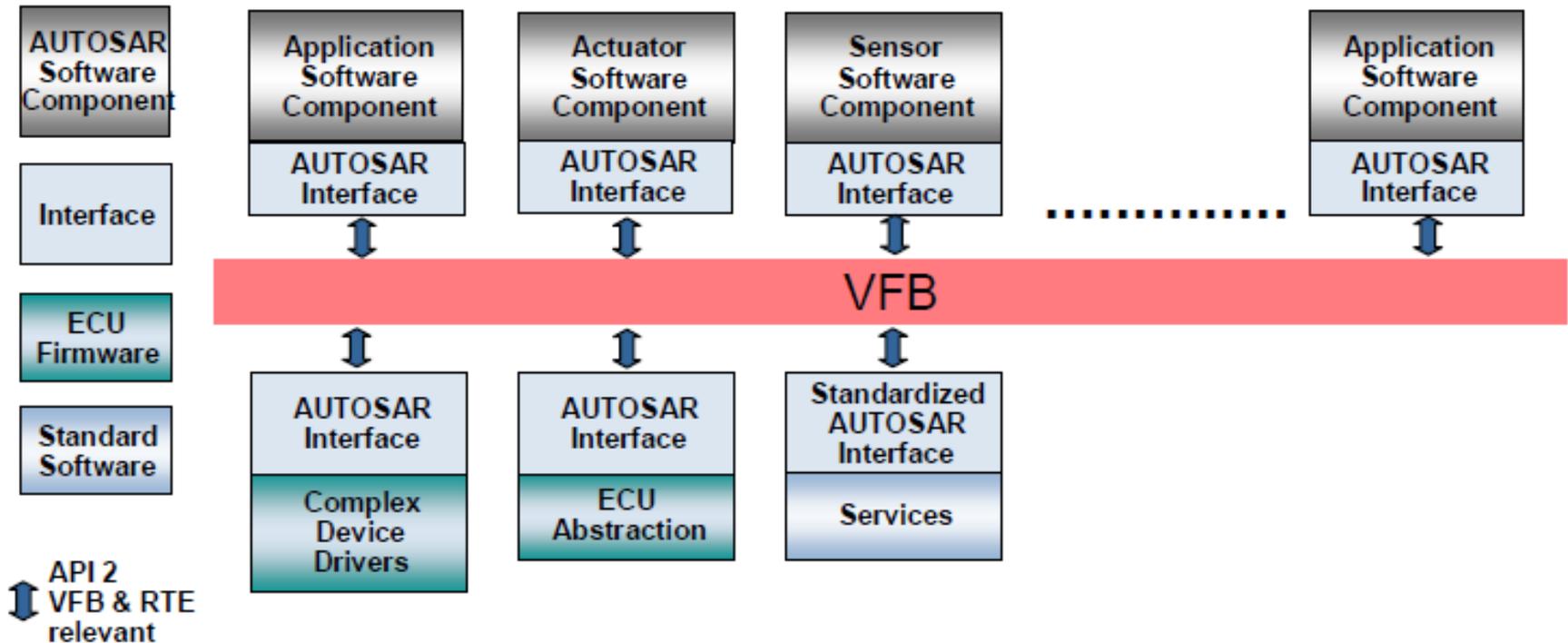
# Outline

- 分层概述
- 应用层
- VFB与RTE层
- 基础软件（BSW）
- 示例



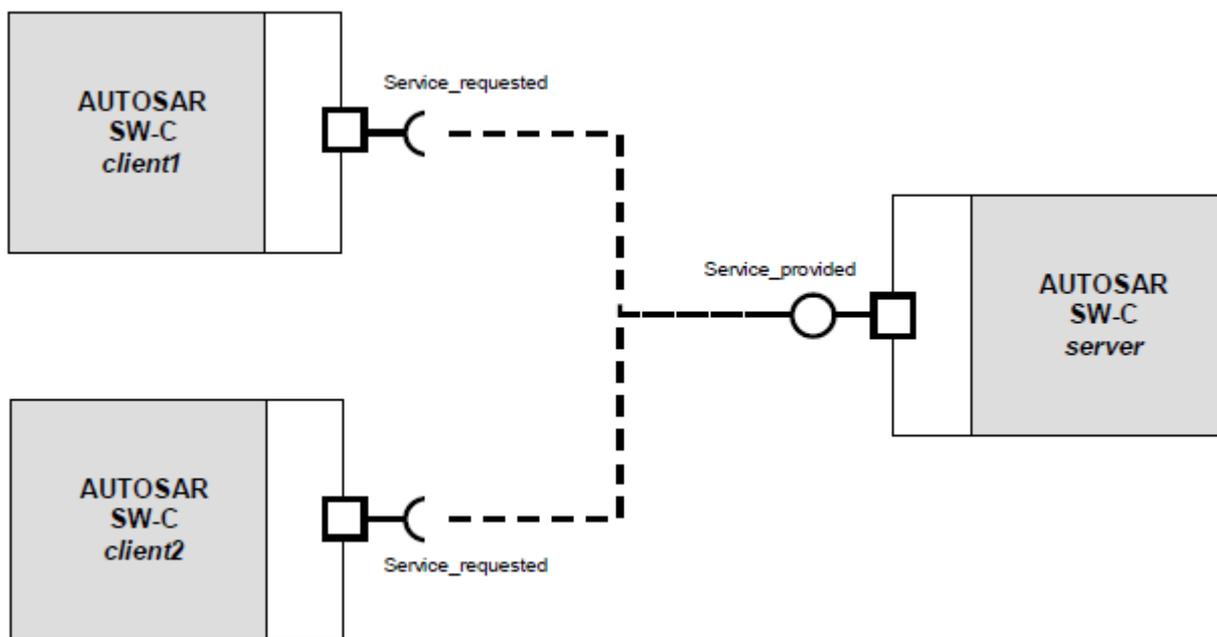
# Virtual Functional Bus

- 所有的Component（包括ASWC、ECU抽象、服务、复杂驱动）之间的通信组成了VFB。



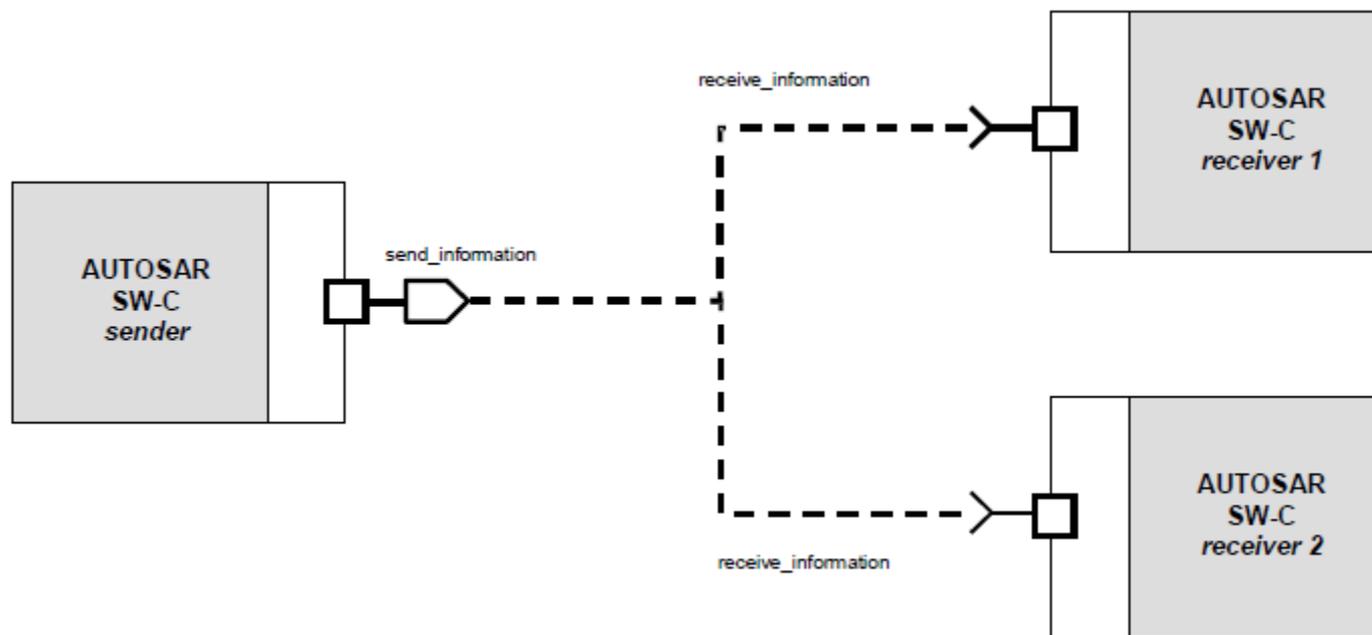
# VFB之Client-Server通信

- 同一ECU内部，CS通信为函数调用
- ECU之间的，CS通信为远程接口调用
- CS通信分为同步和异步通信



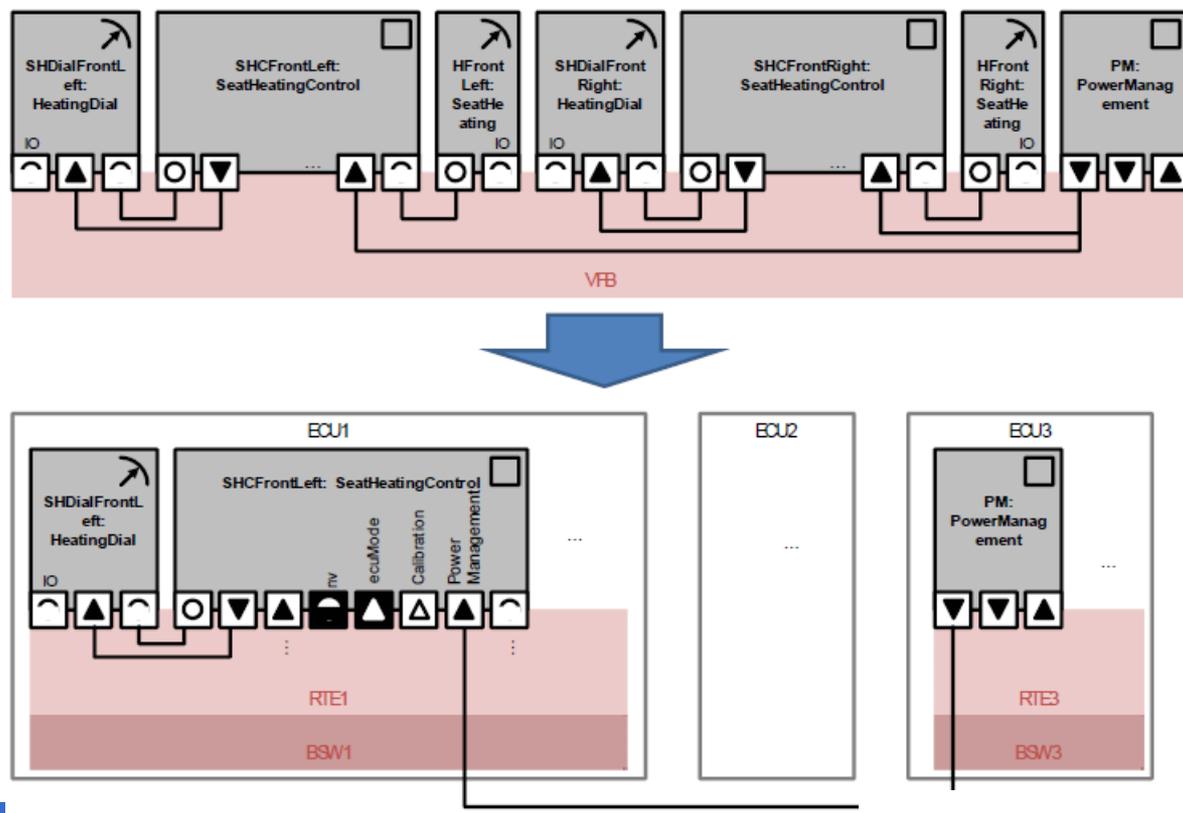
# VFB之Sender-Receiver通信

- 同一ECU内，SR通信为变量传递
- ECU间，SR通信为数据传输
- SR通信为异步的数据传输，Sender端不会收到Receiver的任何响应。



# Runtime Environment

- RTE是VFB在具体一个ECU中的实例。
- RTE实现了应用层SW-C之间、应用层SW-C与BSW之间的具体通信
- RTE通过划分RTOS的任务、资源、事件等，提供给组件一个隔离底层中断的运行环境。



# RTE的通信实现

- SW-C之间的通信是调用RTE API函数而非直接实现的，都在RTE的管理和控制之下。
- 每个API遵循统一的命名规则且只和软件组件自身的描述有关。



Implementation of Client Runnable run1:

```
Rte_Runnable_run1() {  
    ...  
    Rte_Write_wiperWasher_start(...);  
    ...  
}
```

Implementation of WiperWasher Runnable run1:

```
Rte_Runnable_run1() {  
    ...  
    v = Rte_Read_cmd_start(...);  
    ...  
}
```

# RTE的通信实现

- ECU内的SR通信通过变量传递实现

```
Rte_Write_Client_wiperWasher_start(...) {  
    modify variable  
}
```

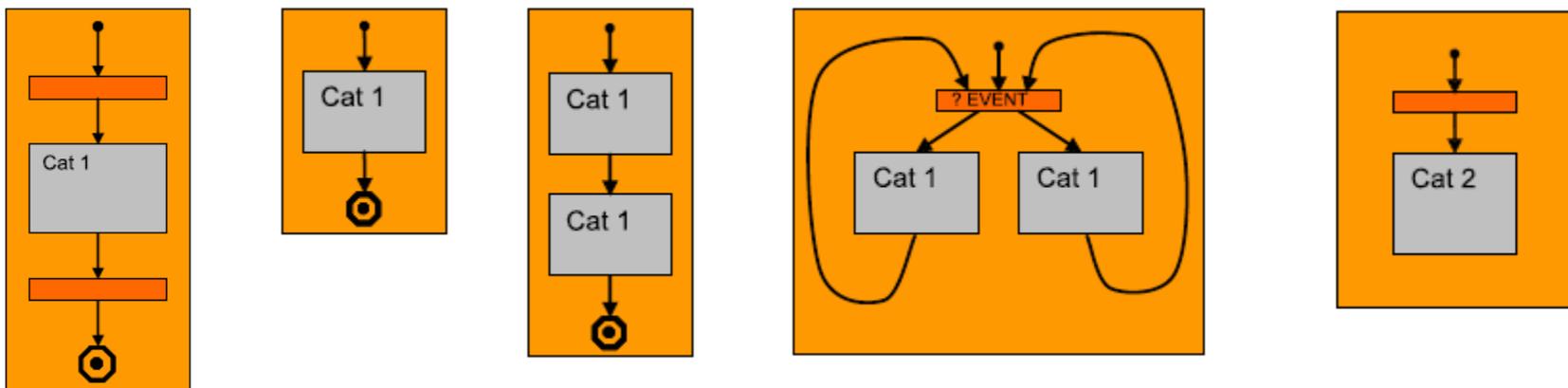
- ECU间的SR通信通过COM服务实现

```
Rte_Write_Client_wiperWasher_start(...) {  
    access COM  
}
```

- 具体通信实现取决于系统设计和配置，都由工具供应商提供的RTE Generator自动生成的。

# 运行实体（Runnable Entity）映射

- SW-C中包含一个或多个运行实体（Runnable Entity）。
- 运行实体映射在RTOS的任务中调度运行，可以分为两类：
  - 无需中间等待的运行实体（Cat 1）：映射为基本任务或扩展任务。
  - 有等待事件的运行实体（Cat 2）：运行实体在运行中间可能需要等待某个事件发生，比如远程调用等待特定返回值。只能将这类运行实体映射为扩展任务，通过事件机制来实现等待同步功能。



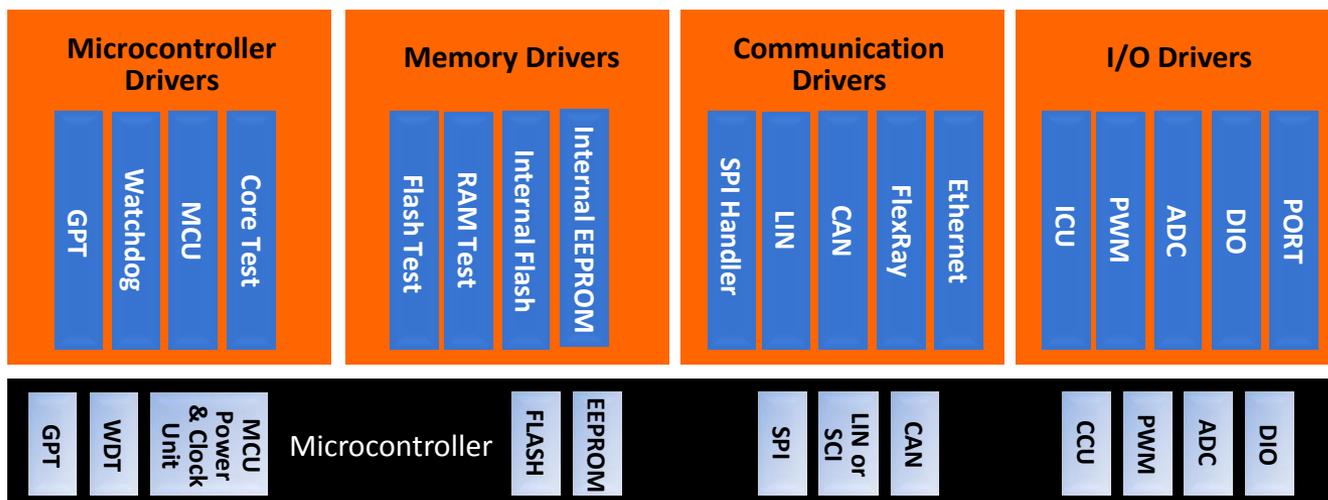
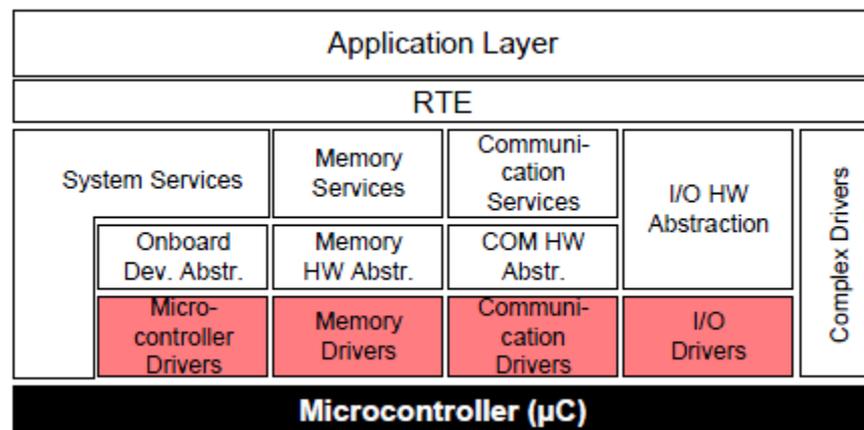
# Outline

- 分层概述
- 应用层
- VFB与RTE层
- 基础软件（BSW）
- 示例



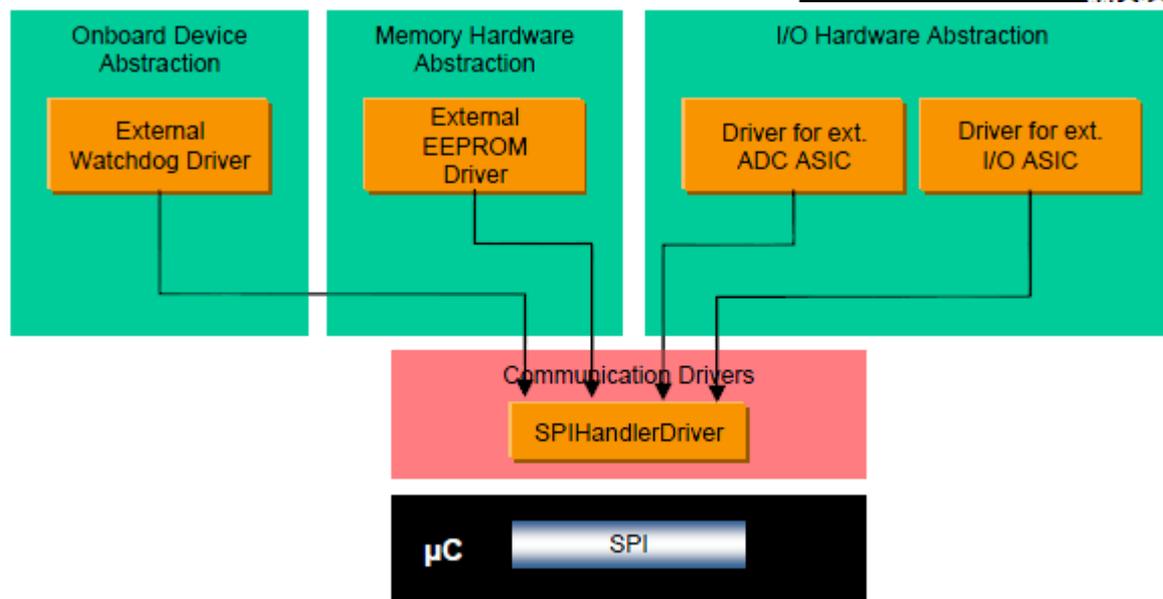
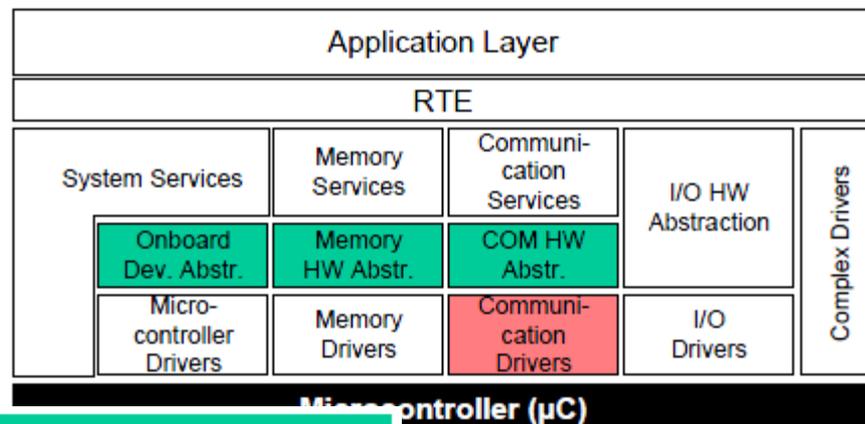
# 微控制器抽象层

- 通信驱动：SPI、CAN等
- I/O驱动：ADC、PWM、DIO等
- 内存驱动：片内EEPROM、Flash等
- 微控制器驱动：看门狗、GPT等



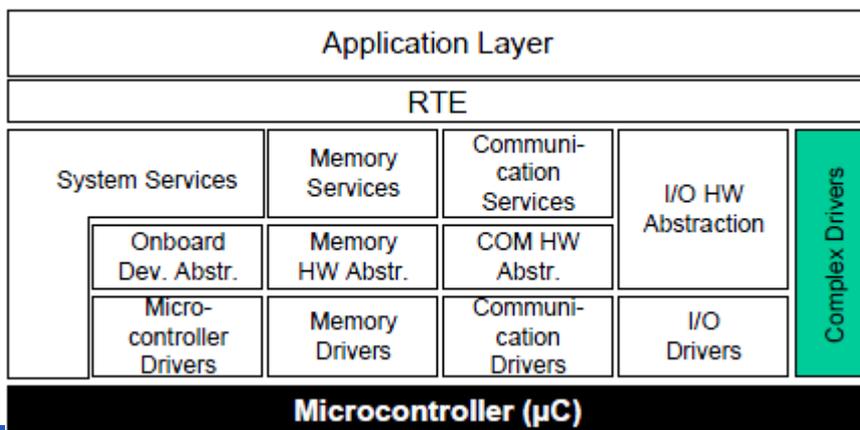
# 微控制器抽象层：SPIHandlerDriver

- SPIHandlerDriver封转了统一访问SPI总线的接口，上层软件可以并发的多个访问。如下图

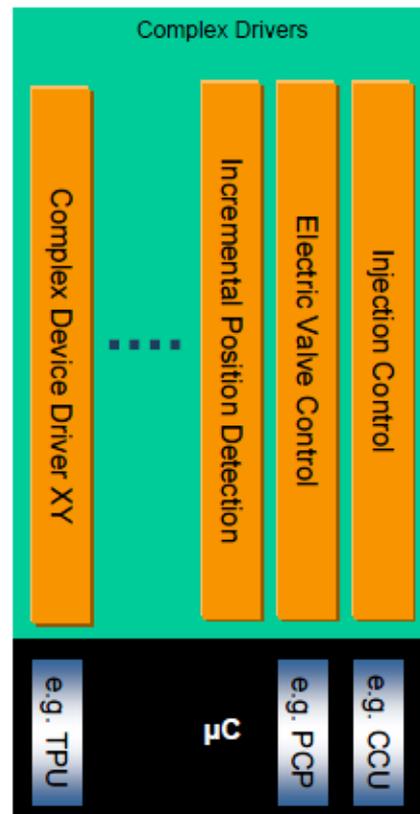


# 复杂驱动

- 利用中断、TPU、PCP等，复杂驱动可以实现了实时性高的传感器采样、执行器控制等功能：比如
  - Injection Control
  - Electric Valve Control
  - Incremental Position Detection

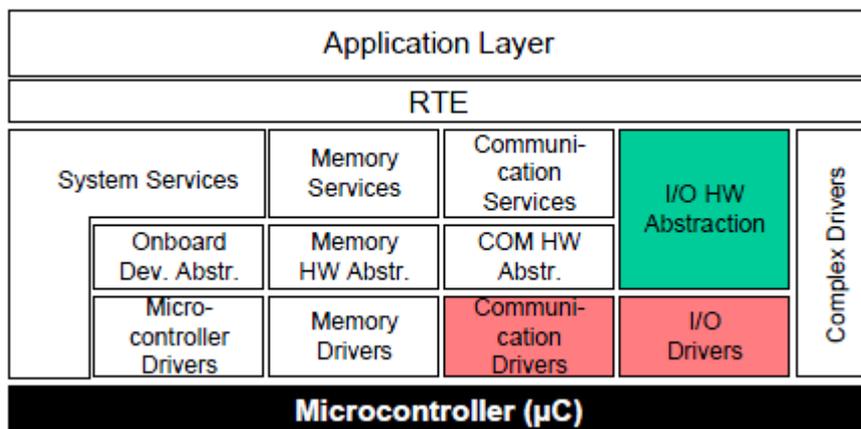


例如：

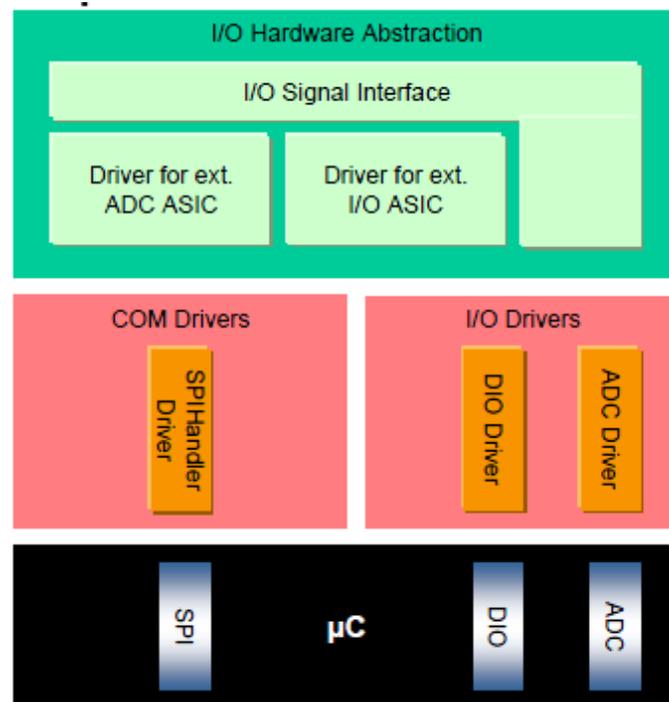


# I/O硬件抽象

- 可以通过I/O硬件抽象中的信号接口来访问不同的I/O设备
- 将I/O信号都进行了封装，比如电流、电压等

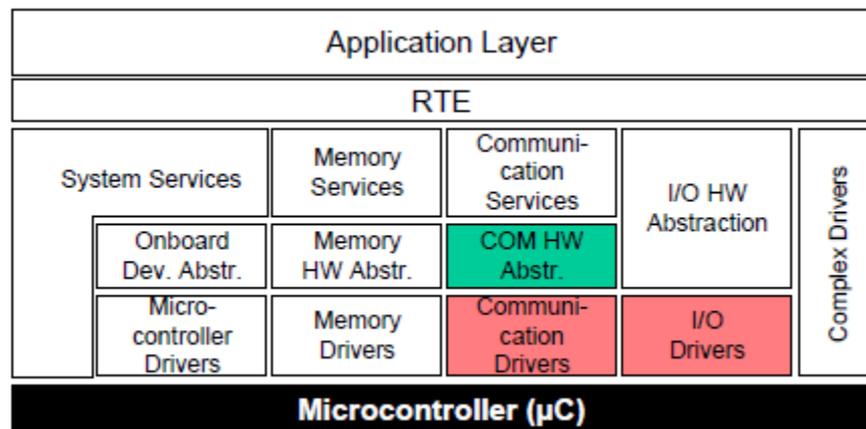


例如：

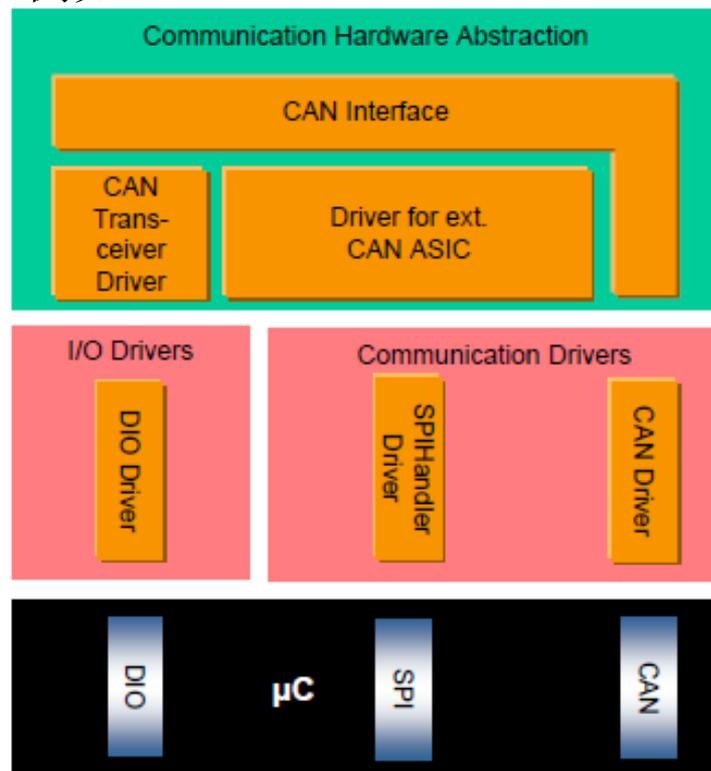


# COM硬件抽象

- COM硬件抽象将微控制器、板上的所有通信通道进行了封装。
- 将LIN、CAN、FlexRay等通信方式都进行了抽象定义

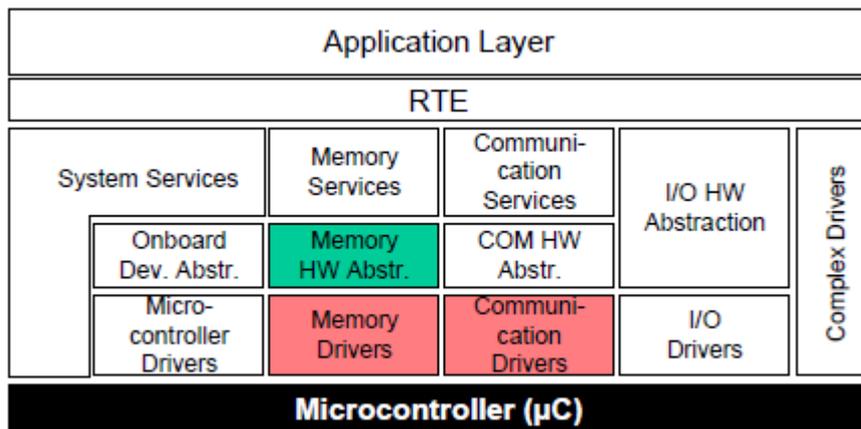


例如：

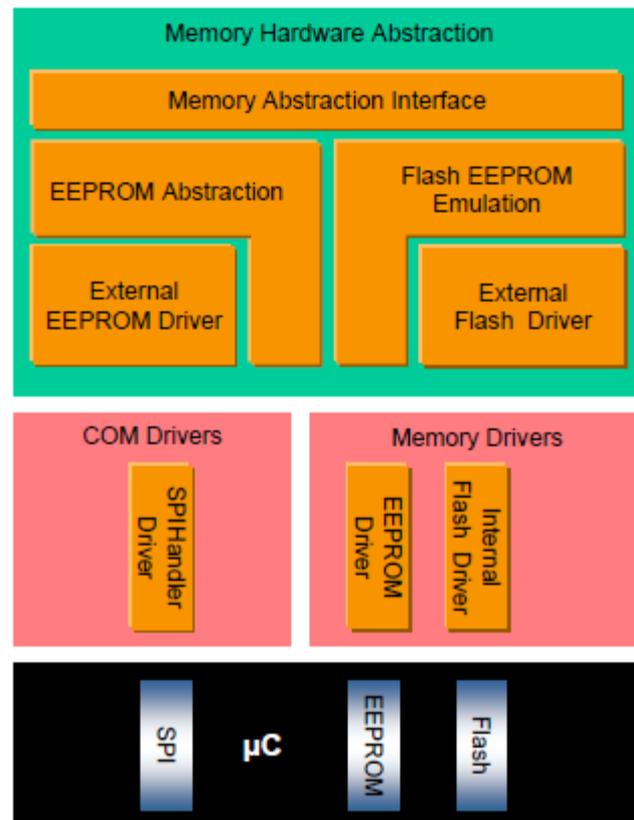


# Memory硬件抽象

- 将片内、板上的内存资源进行了统一封装，比如对片内EEPROM和片外的EEPROM都提供了统一的访问机制



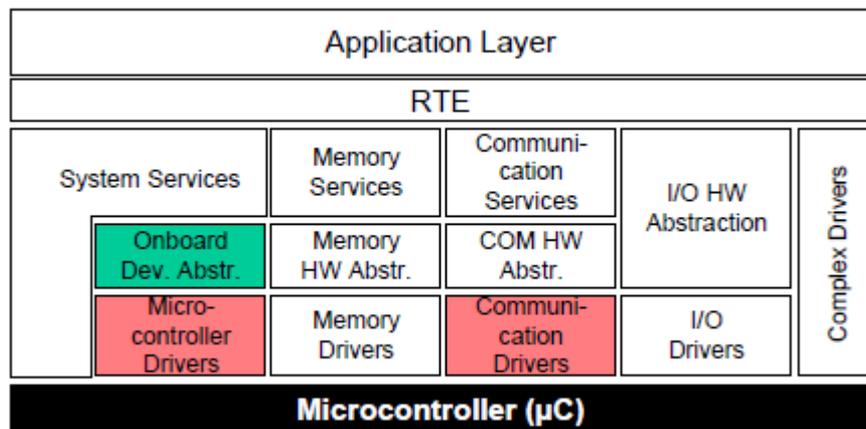
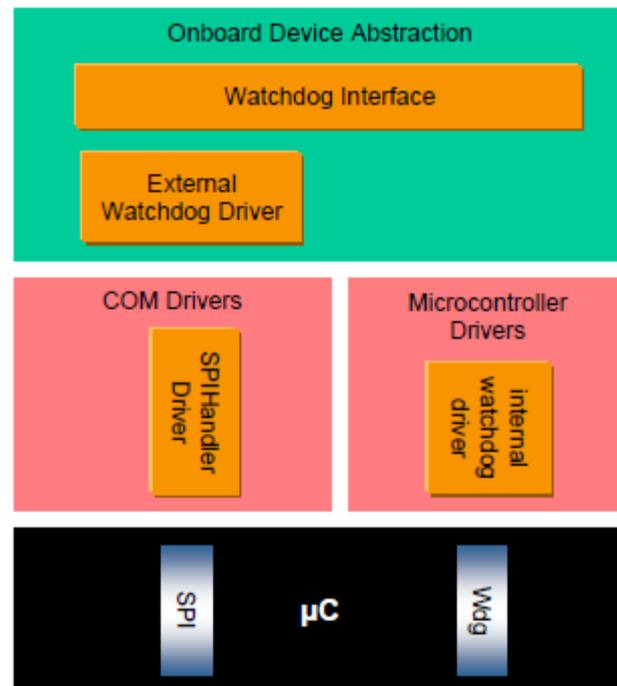
## Example:



# Onboard Device Abstraction

- 将ECU硬件上特殊的外设（即不是用于传感，也不用于执行的）进行封装，比如 Watchdog。

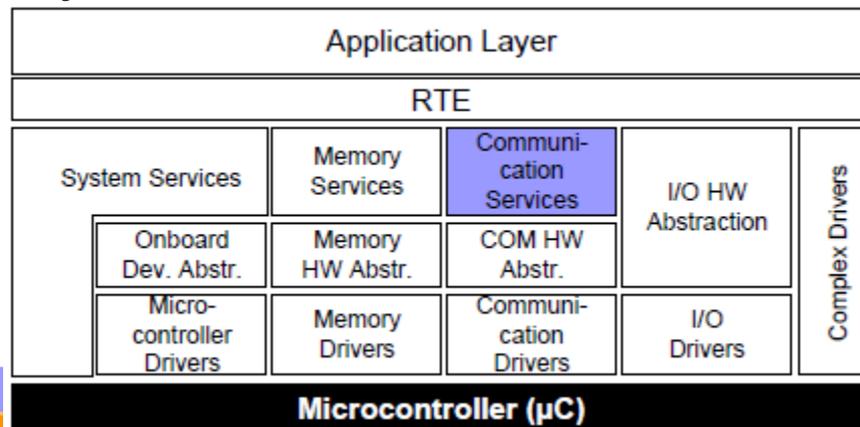
Example:



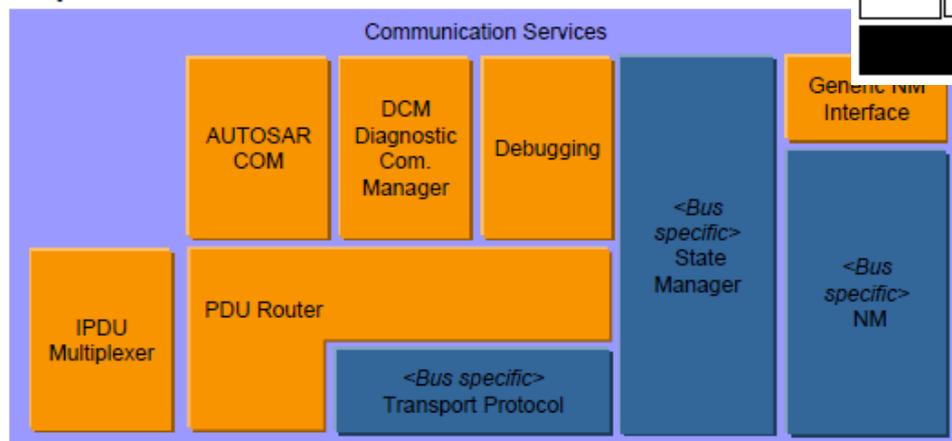
# 通信服务

## ■ 通信服务封转了CAN、LIN、FlexRay等通信接口：

- 提供统一的总线通信接口
- 提供统一的网络管理服务
- 提供统一的诊断通信接口



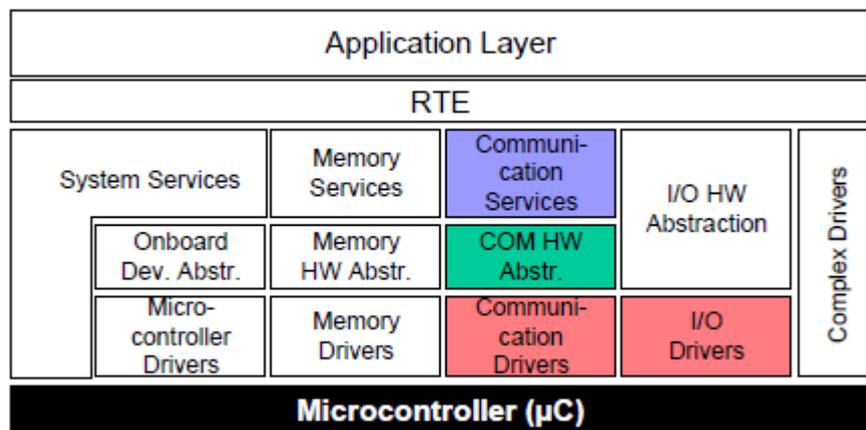
### Example:



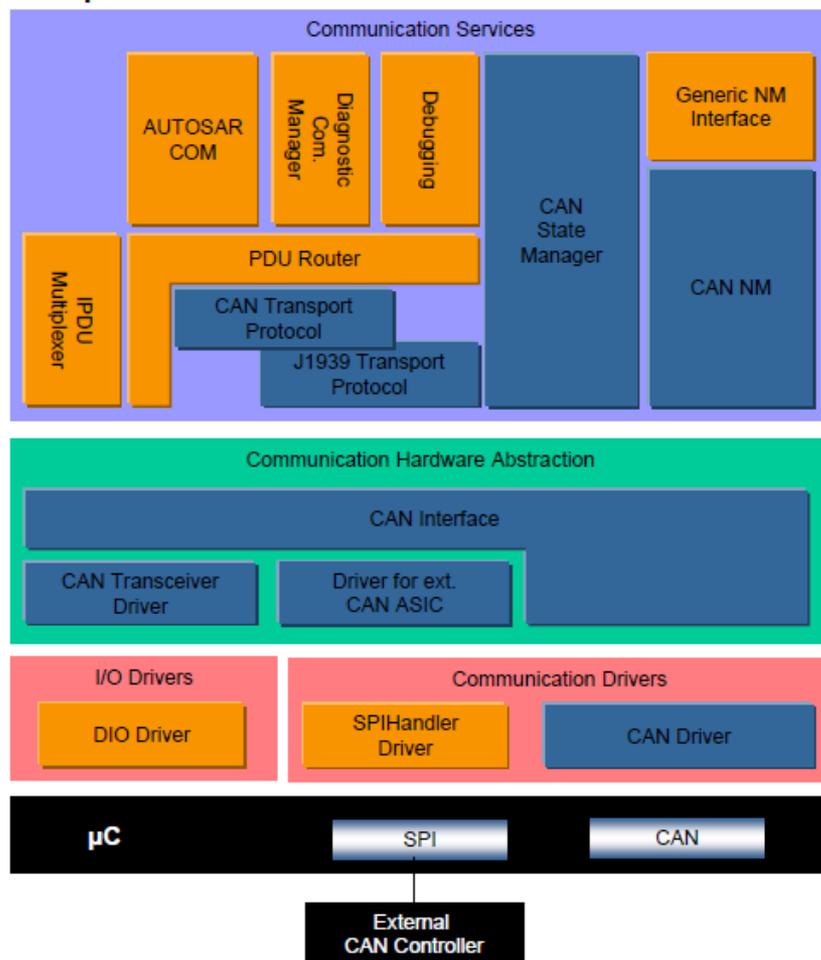
■ Bus specific modules

# 通信服务-CAN

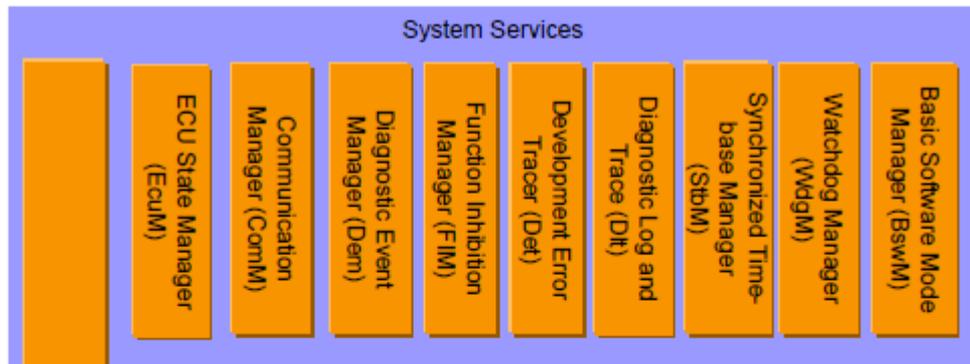
- 针对CAN的通信服务封装了具体的协议、消息属性，提供了统一的接口供应用层调用。
- 两种传输协议：J1939 TP、CanTP



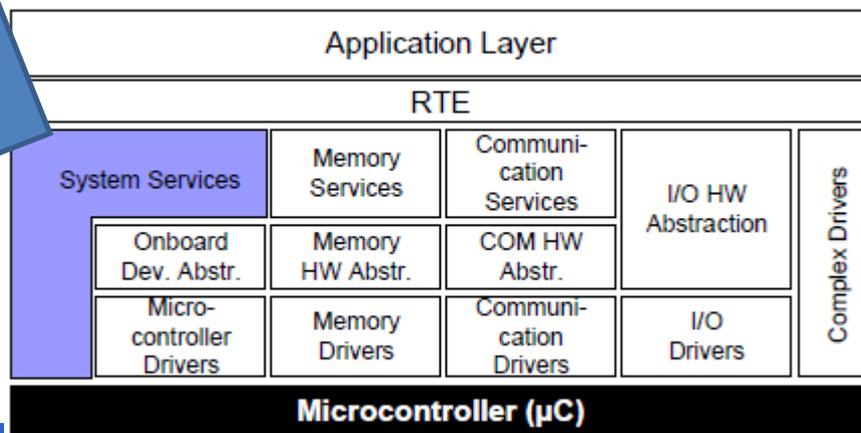
Example:



# 系统服务

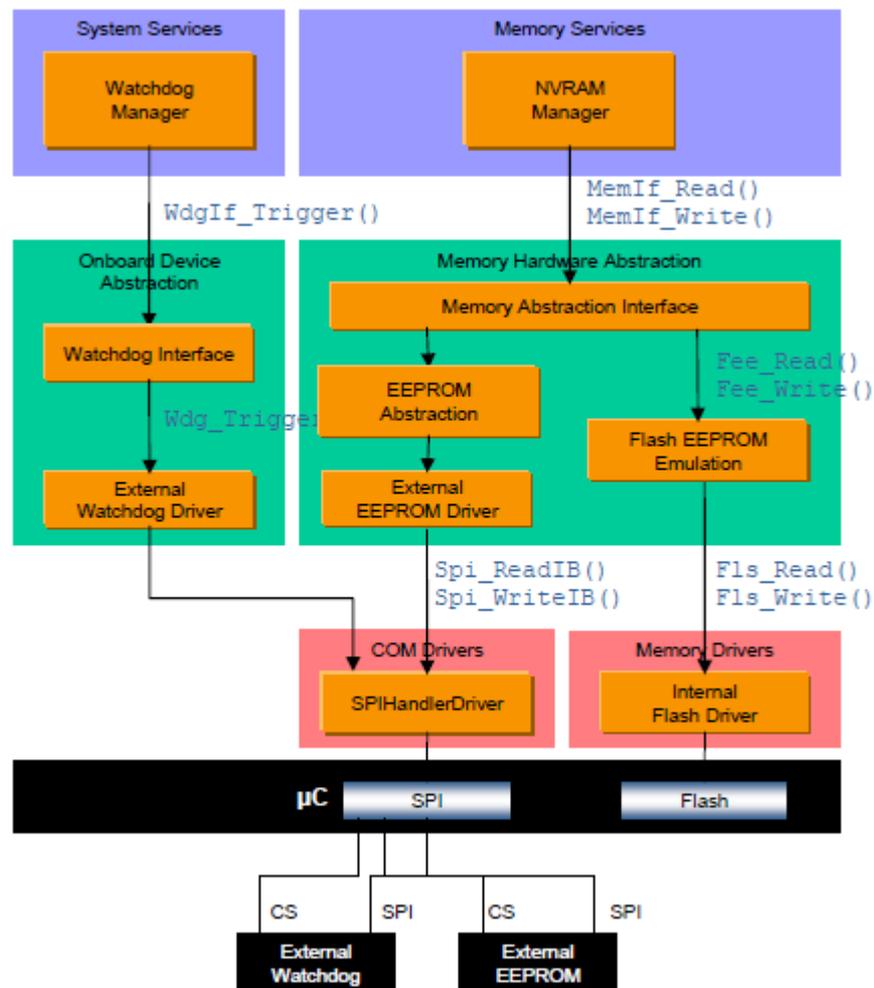


- 提供RTOS服务，包括中断管理、资源管理、任务管理等
- 提供功能禁止管理、通信管理、ECU状态管理、看门狗管理、同步时钟管理、基本软件模式管理等服务



# BSW示例

- 假设硬件条件如下：
  - ECU包含外部EEPROM和外部Watchdog，都通过SPI来与微控制器连接；
  - SPIHandlerDriver控制SPI的并发访问，其中EEPROM的访问优先级更高；
  - 微控制器也含有EEPROM，可以与外部EEPROM同步使用。



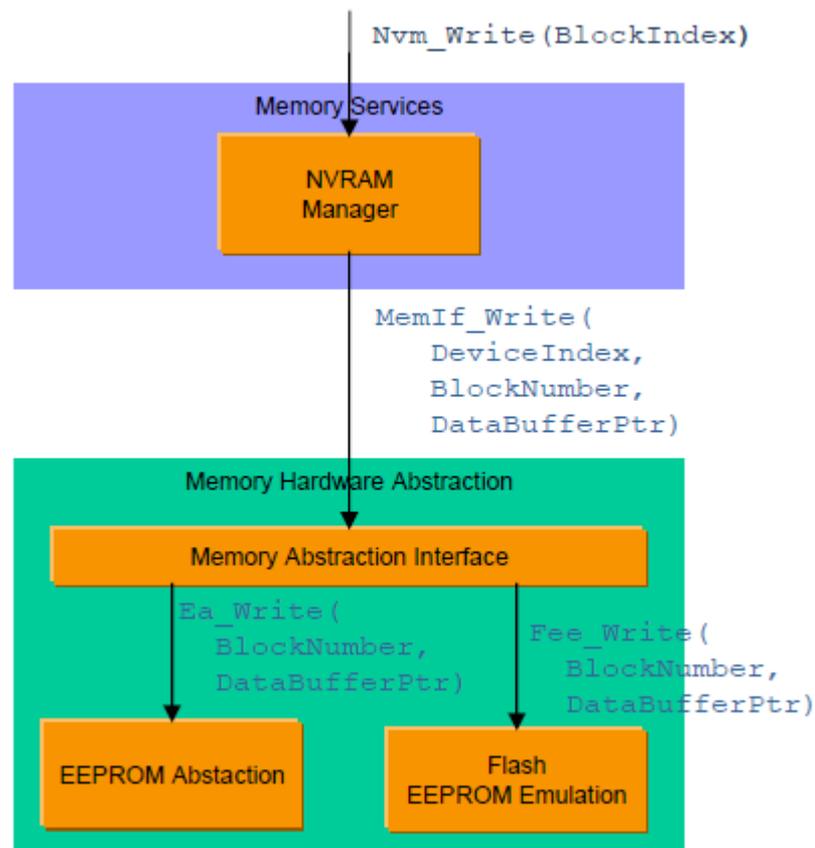
# BSW示例-Memory Service概述

- Memory Service中的NVRAM Manager通过DeviceIndex来区分不同的内存硬件。
- 调用Memory Abstraction Interface中的以下接口来实现写操作：

```
MemIf_Write(DeviceIndex,BlockNumber,DataBufferPtr);
```

- 上述接口的具体实现可以通过调用EEPROM Abstraction或Flash EEPROM Emulation的接口来实

现



# BSW示例-Memory硬件抽象的实现

- 情景1：只有一个EEPROM使用时
  - 文件 MemIf.h

```
#include "Ea.h"          /* for providing access to the EEPROM Abstraction */  
...  
#define MemIf_Write(DeviceIndex, BlockNumber, DataBufferPtr) \  
    Ea_Write(BlockNumber, DataBufferPtr)
```

- 文件MemIf.c: 不需要，因为直接通过宏定义就完成了MemIf\_Write的实现。

# BSW示例-Memory硬件抽象的实现（续）

- 情形2：2个或2个以上不同的内存硬件使用时：

- 文件 MemIf.h：将MemIf\_Write通过DeviceIndex在指针数组中映射

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

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

- 文件 Memif.c：定义一个函数指针的数组

```
#include "Ea.h"          /* for getting the API function addresses */  
#include "Fee.h"        /* for getting the API function addresses */  
#include "MemIf.h"      /* for getting the WriteFctPtrType          */
```

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

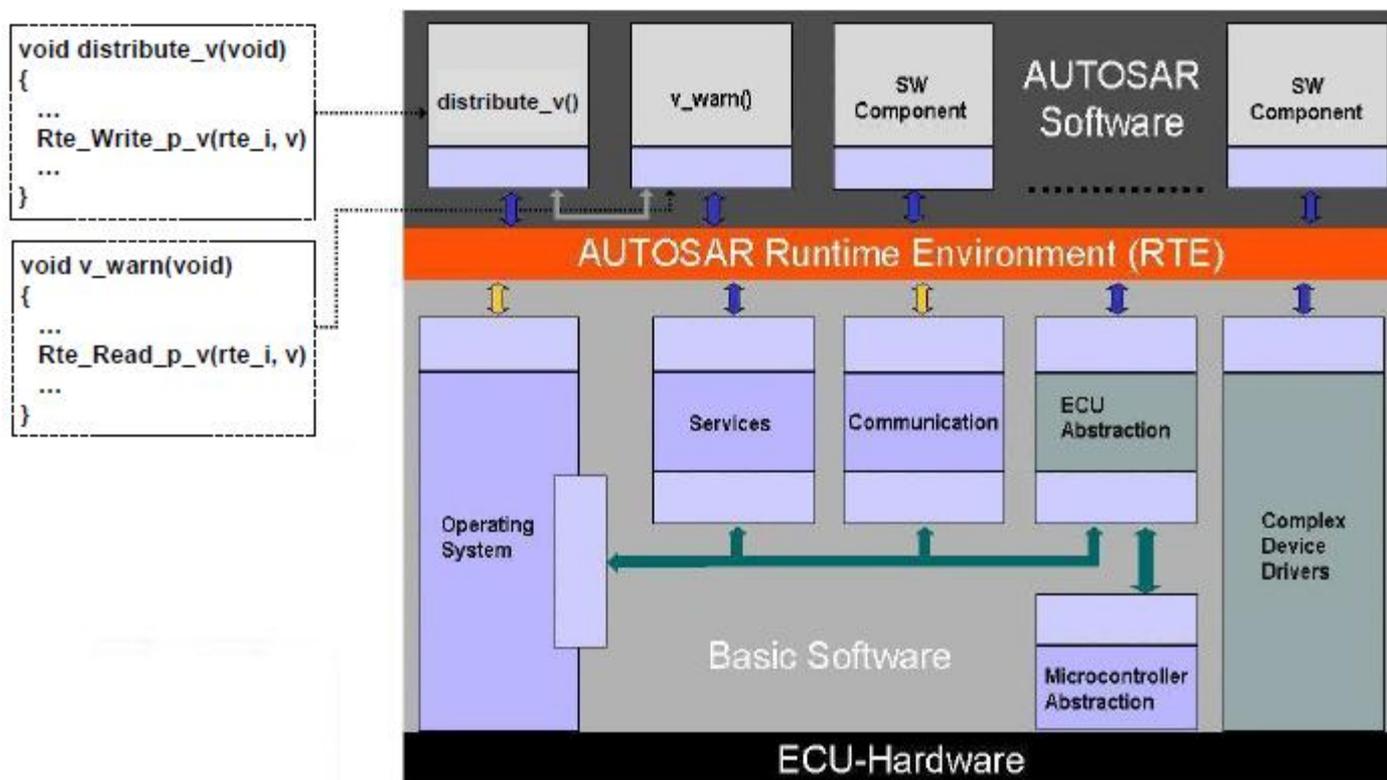
# Outline

- 分层概述
- 应用层
- VFB与RTE层
- 基础软件（BSW）
- 示例



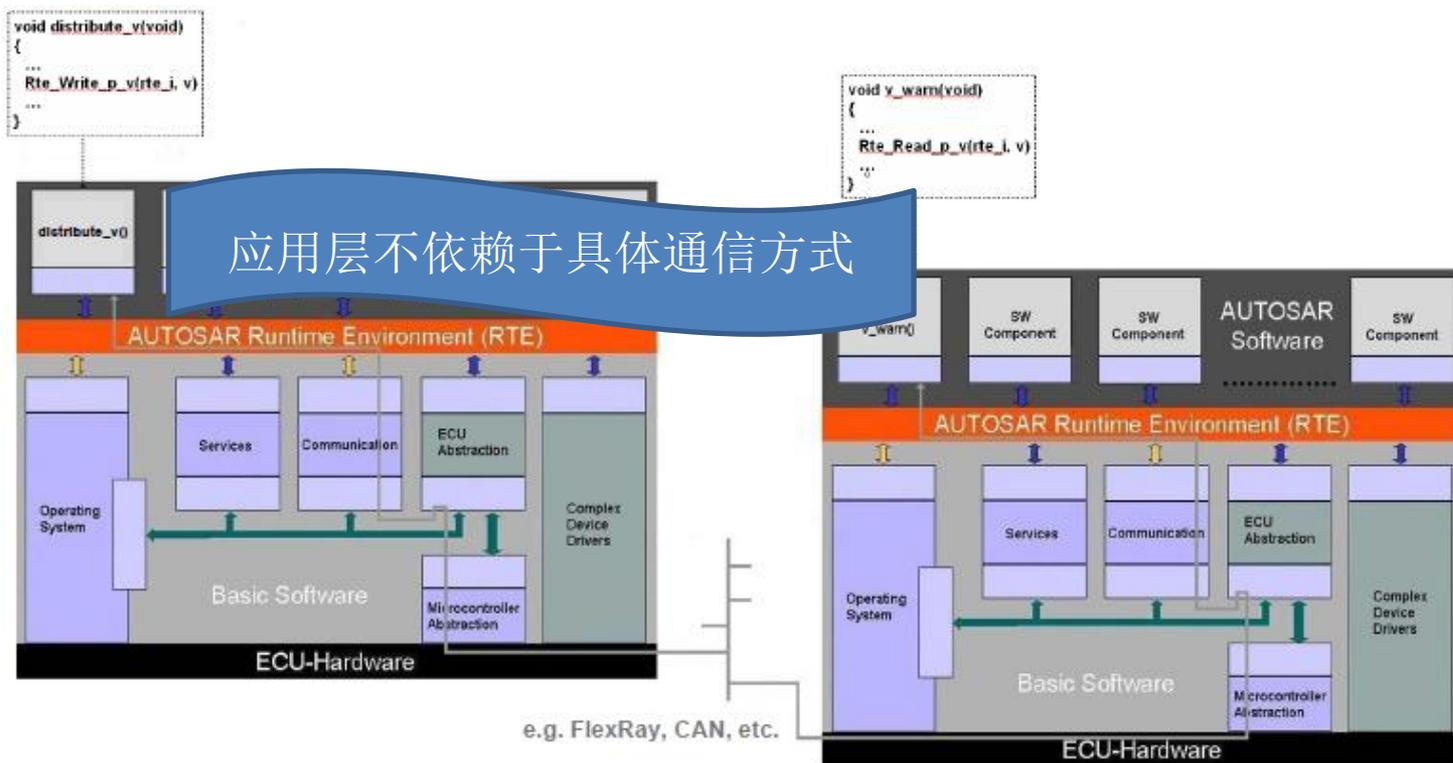
# 示例1：Pedal Management

- 同一ECU内进行通信，应用层各Component的代码如下图所示



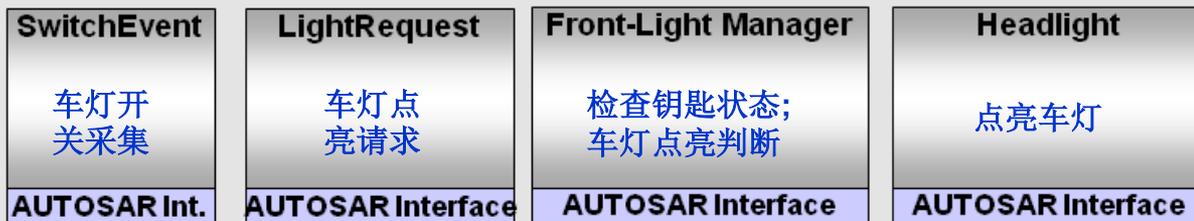
# 示例1：Pedal Management

- ECU间进行通信，应用层各Component的代码与上图中的一致。改变的是RTE这一层的实现。

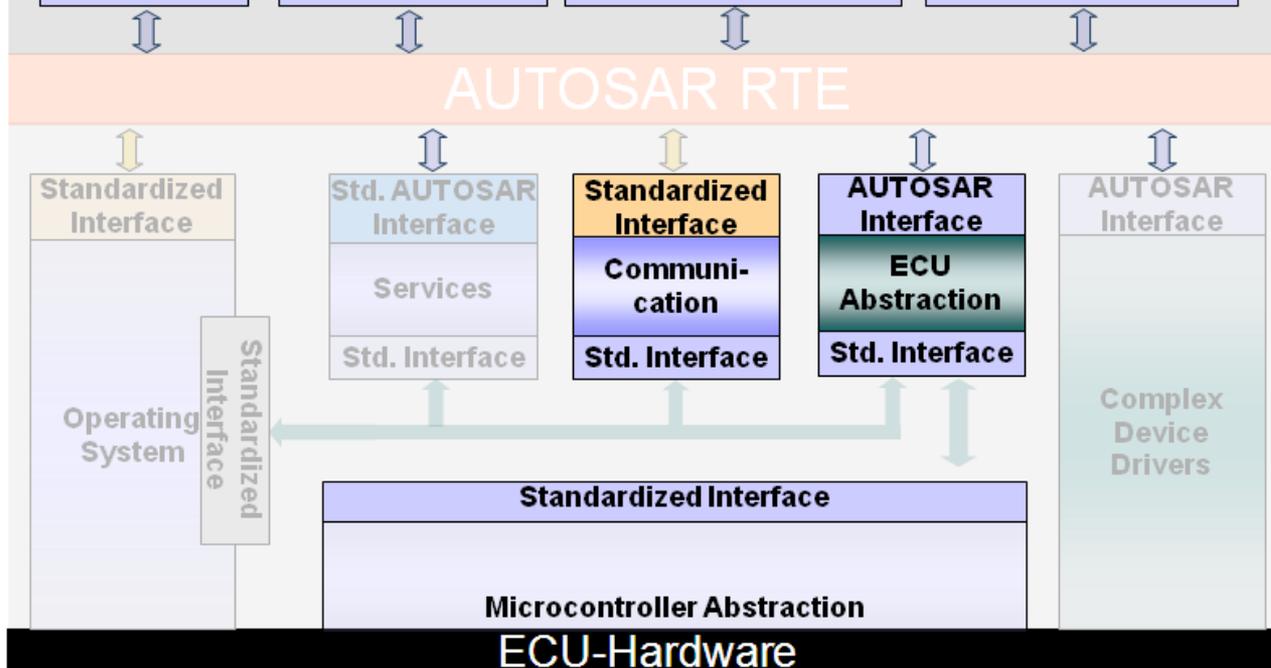


# 示例2：前车灯管理系统

功能架构

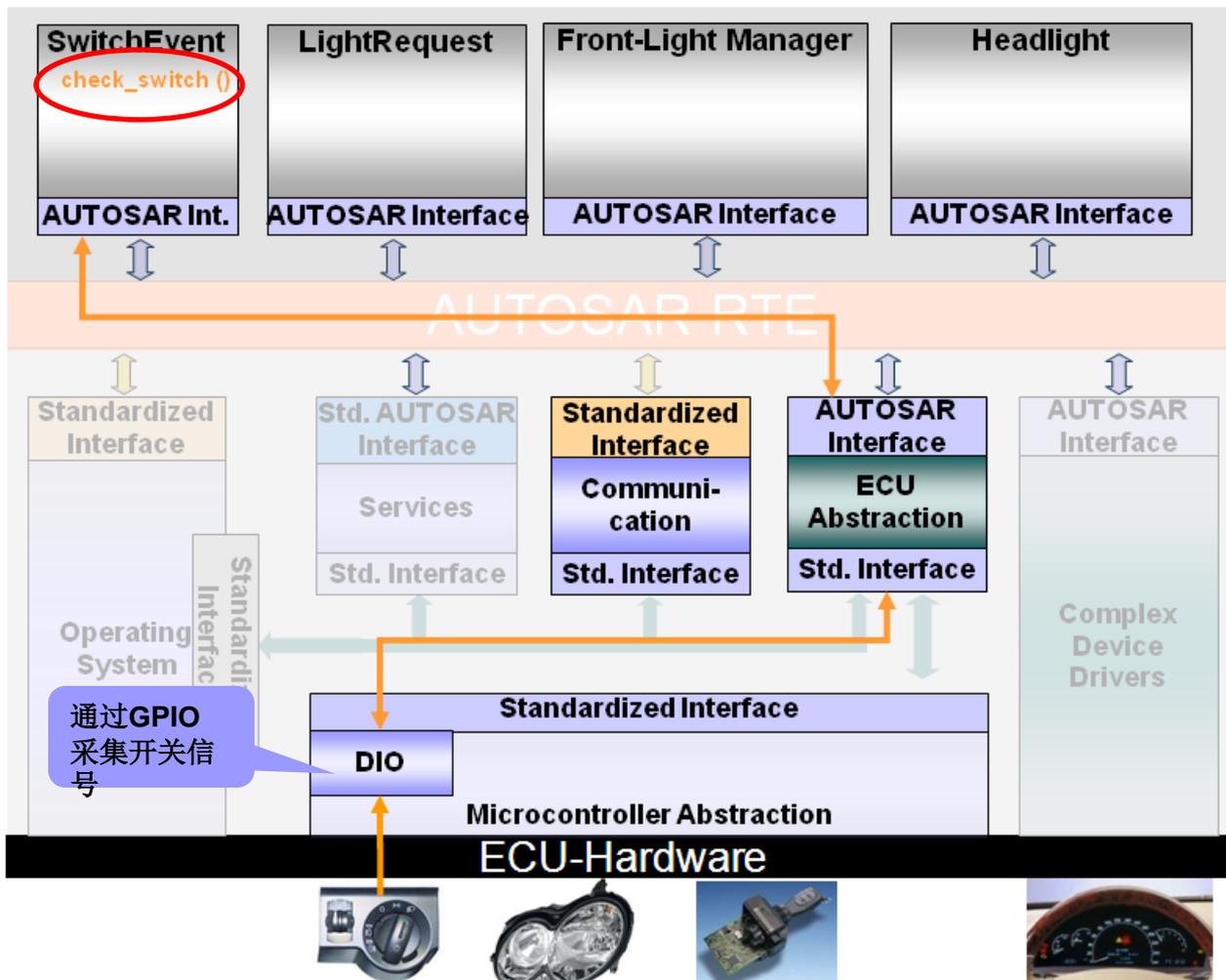


物理架构



# 示例2：前车灯管理系统

功能架构

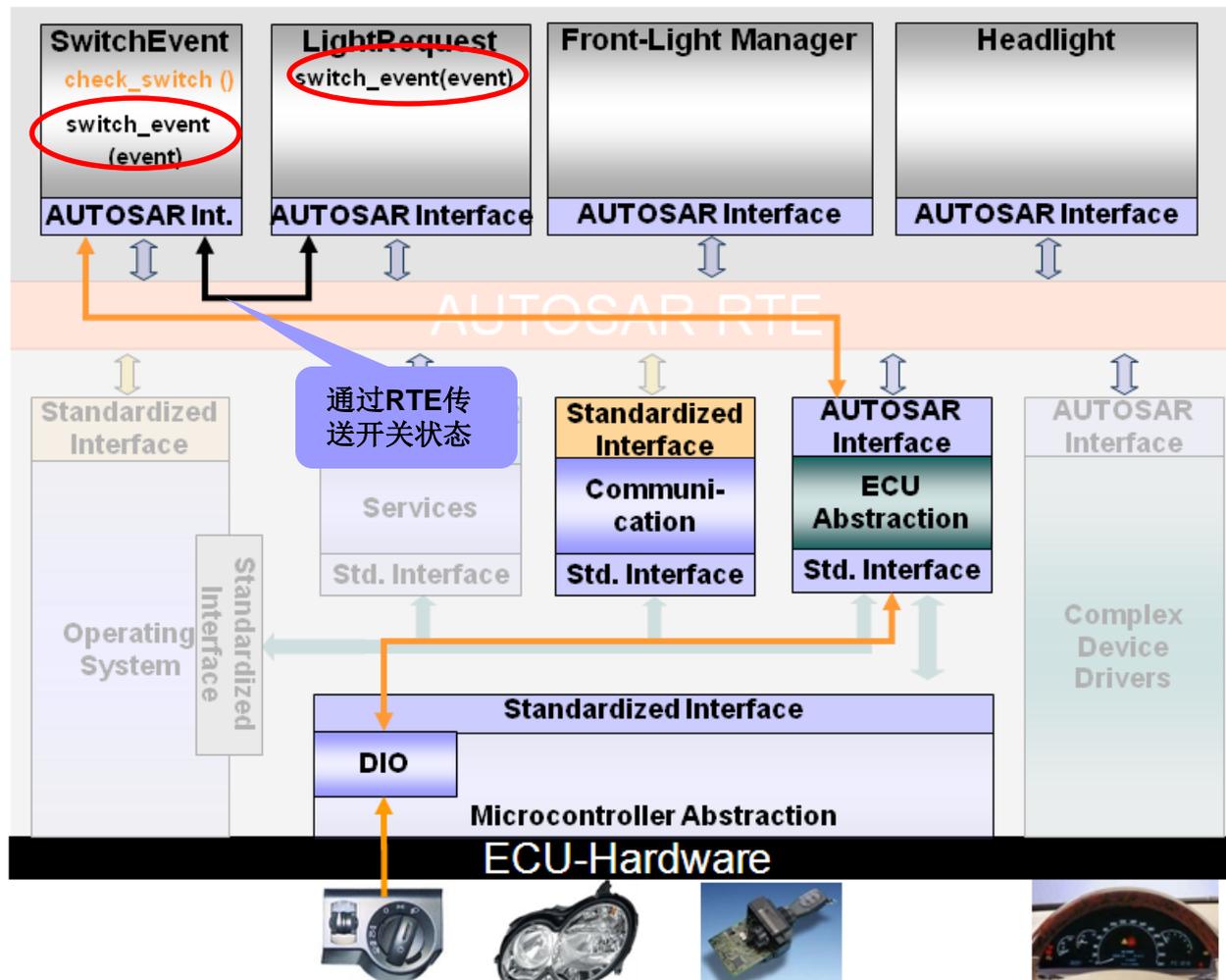


物理架构

# 示例2：前车灯管理系统

功能架构

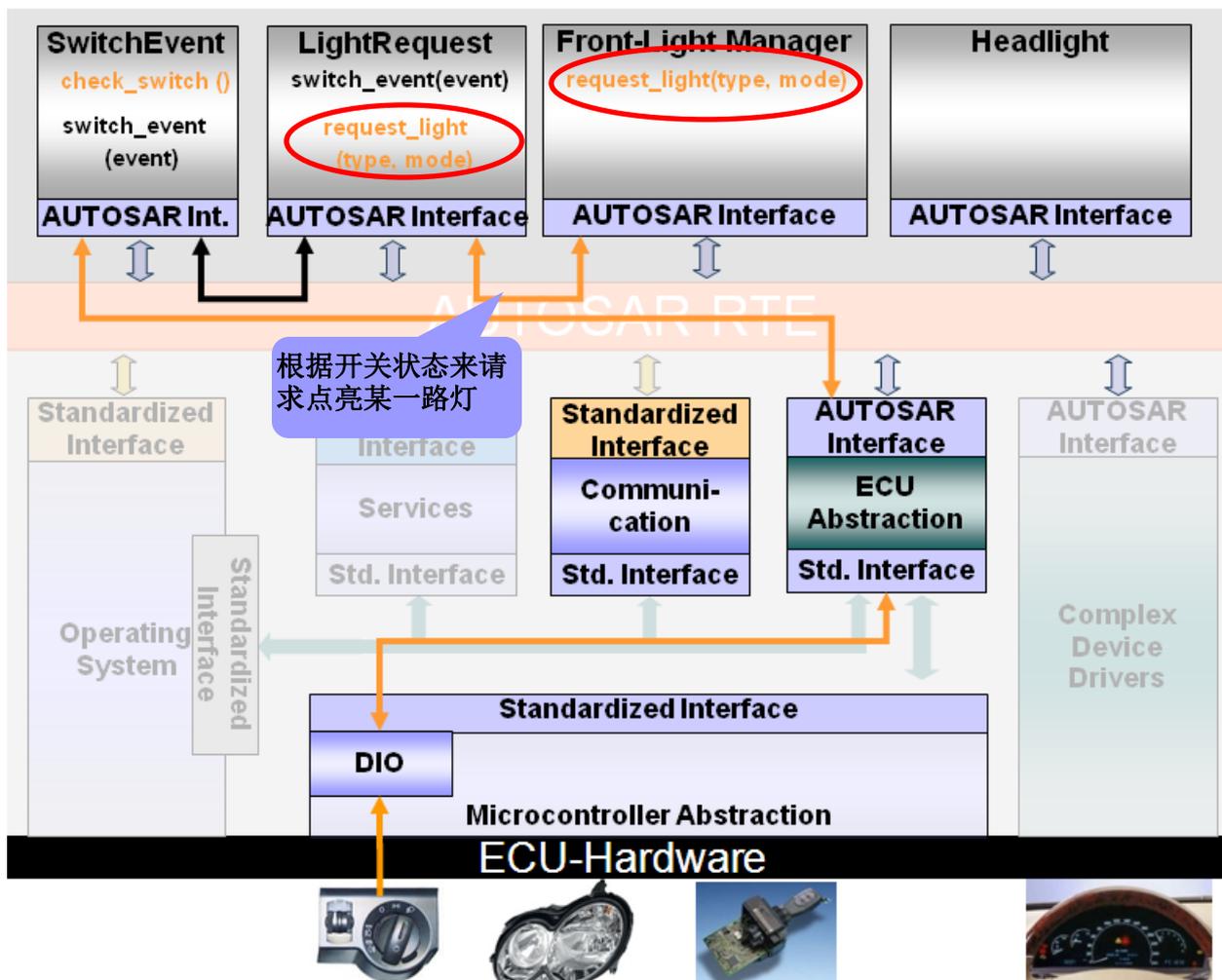
物理架构



# 示例2：前车灯管理系统

功能架构

物理架构

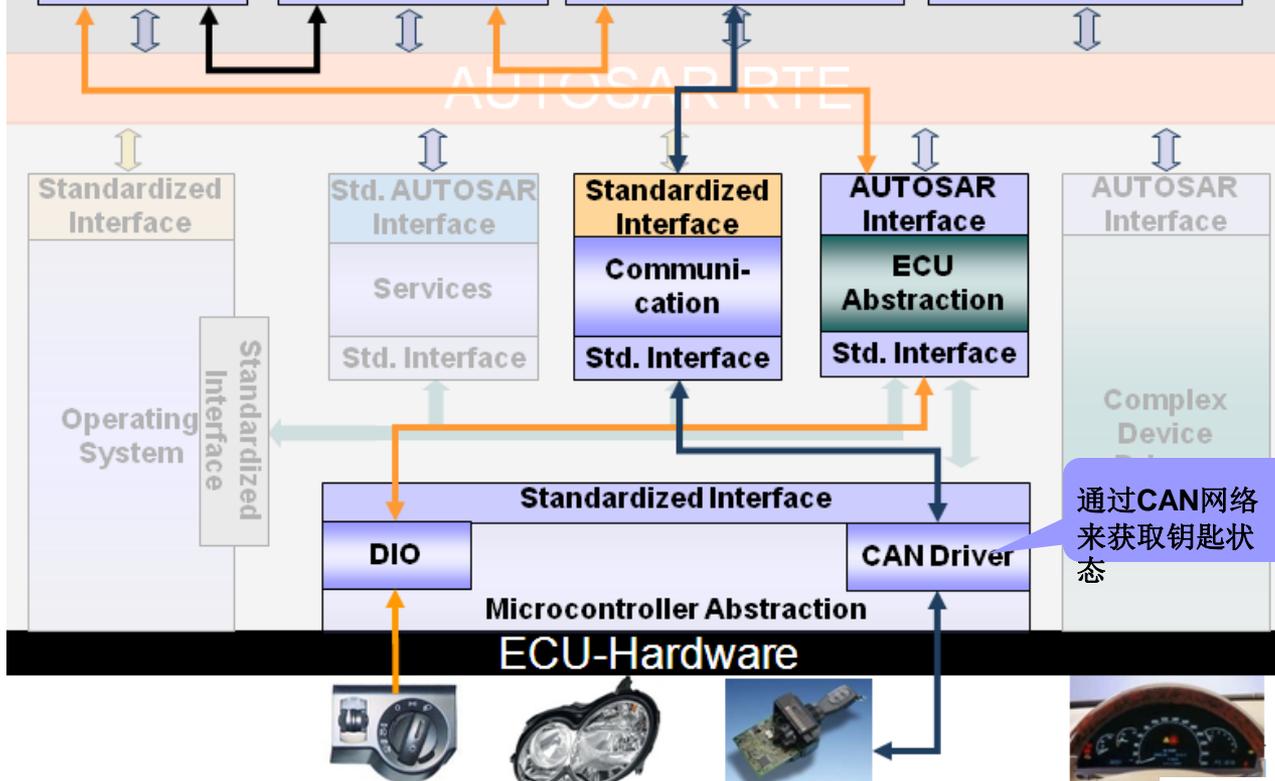


# 示例2：前车灯管理系统

功能架构



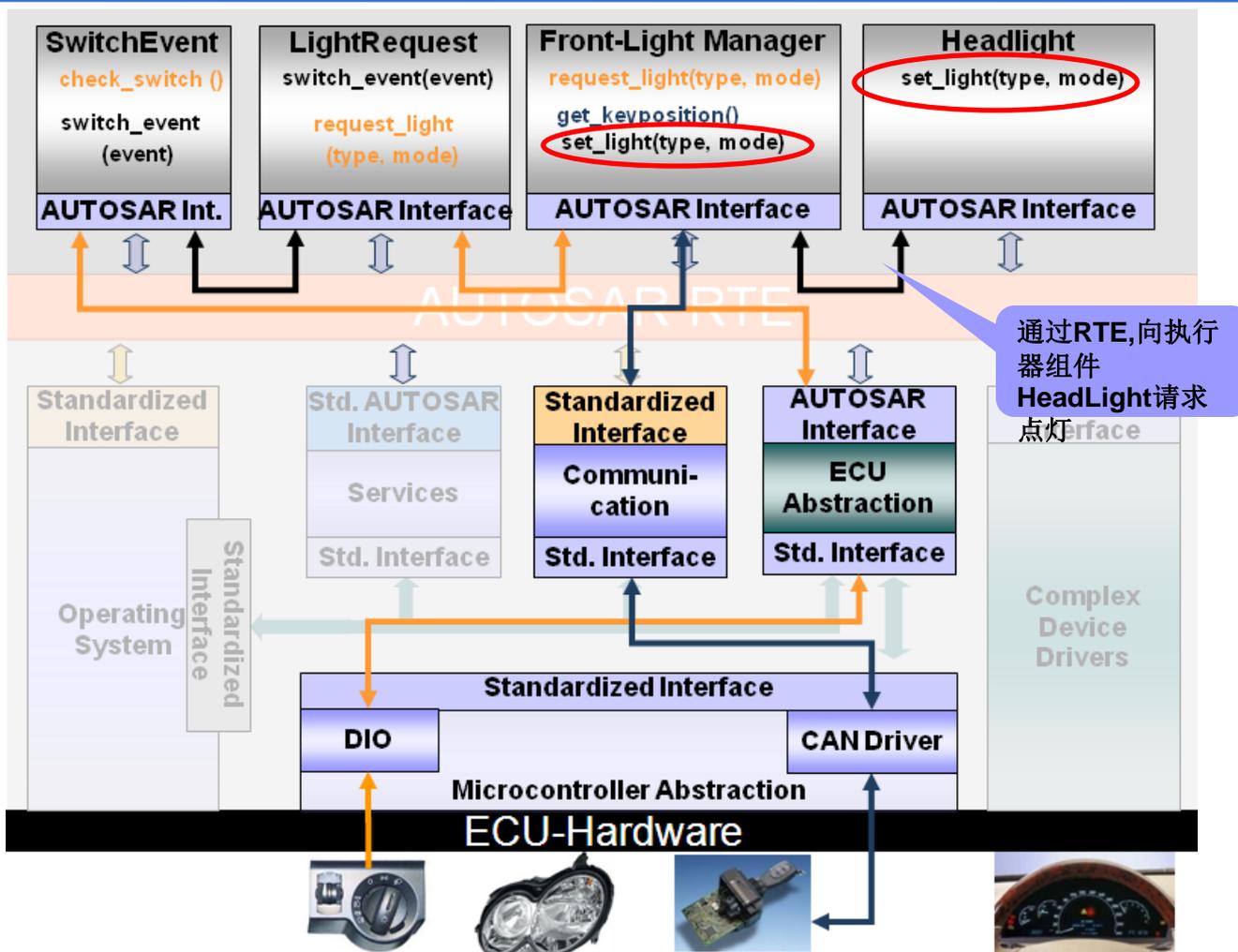
物理架构



# 示例2：前车灯管理系统

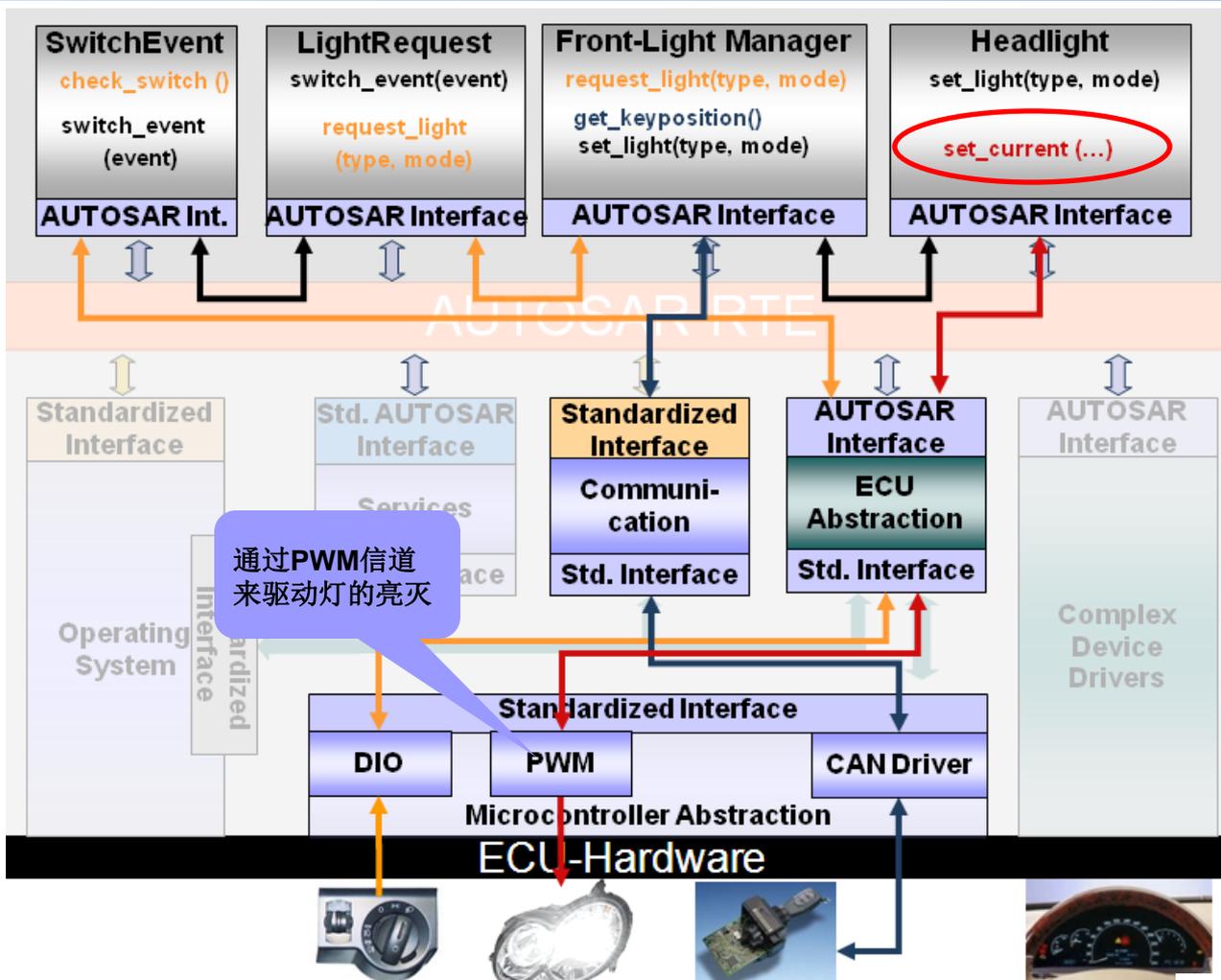
功能架构

物理架构



# 示例2：前车灯管理系统

功能架构

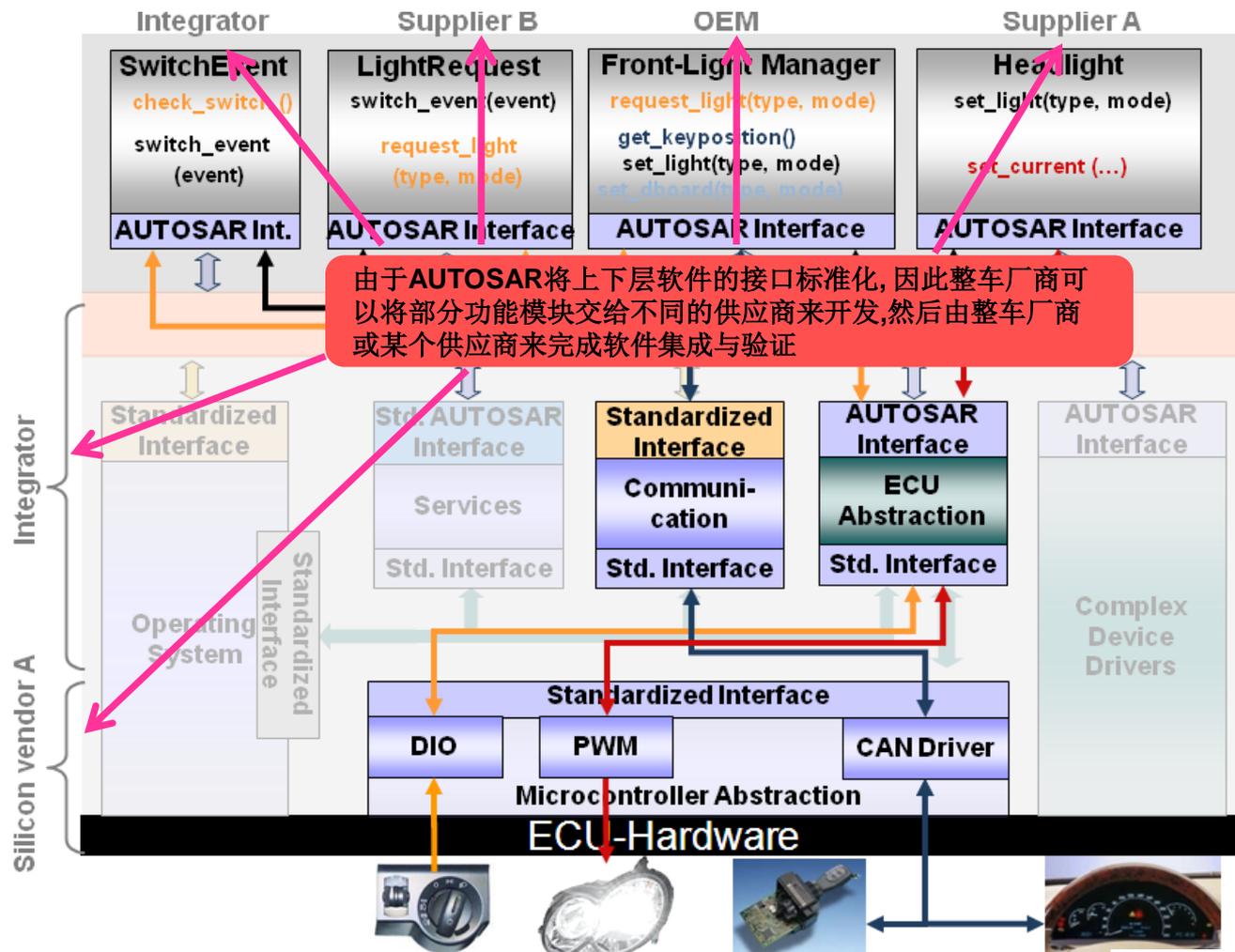


物理架构

# 示例2：前车灯管理系统

功能架构

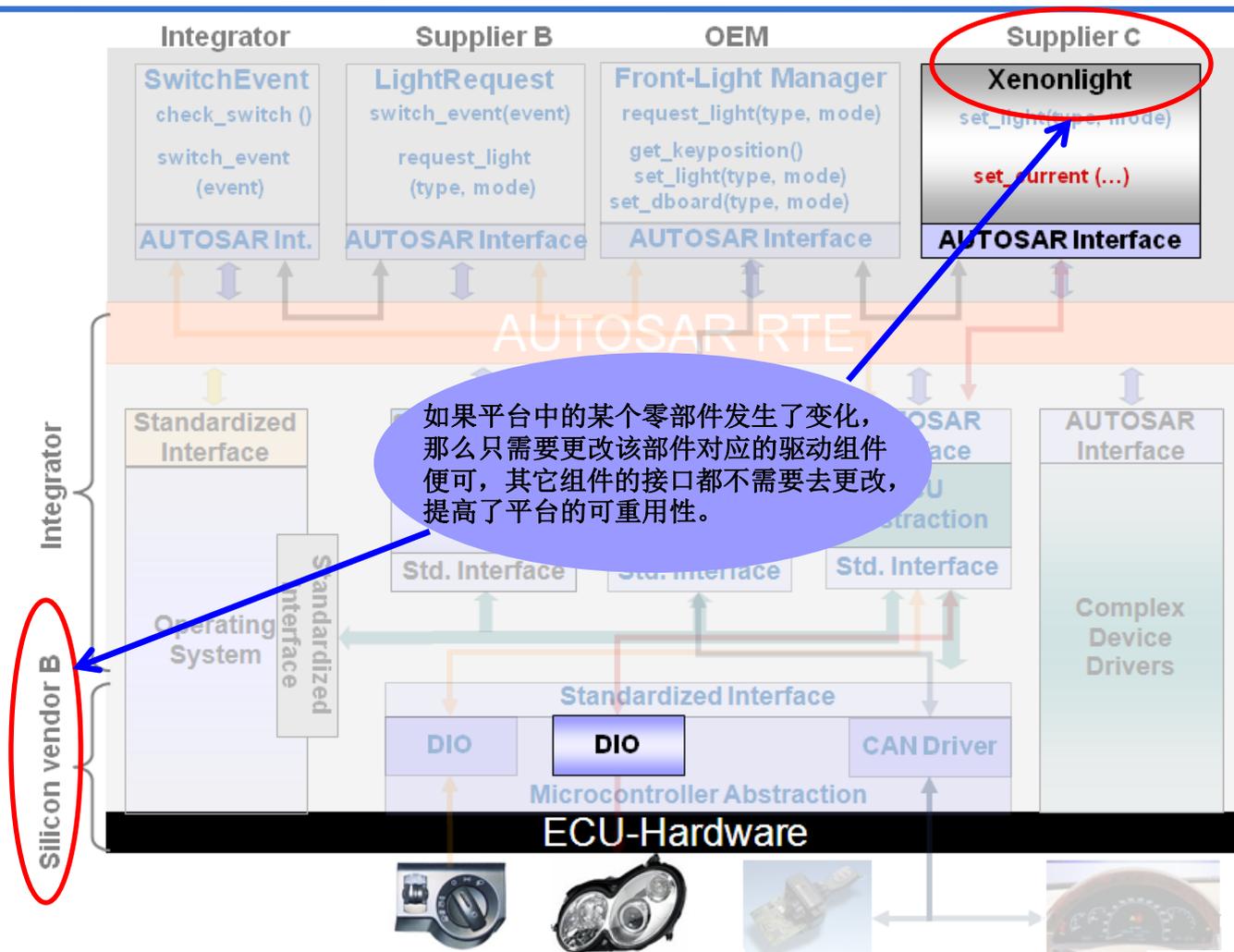
物理架构



# 示例2：前车灯管理系统

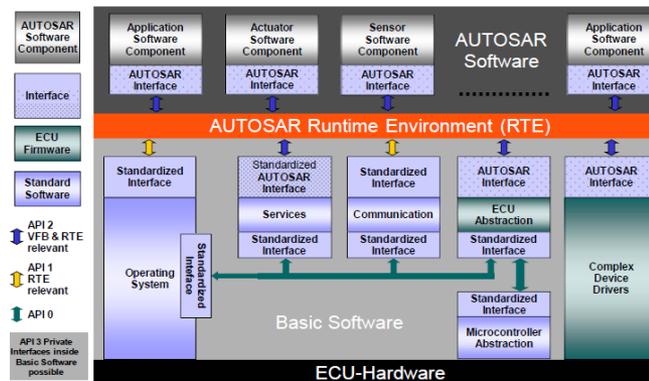
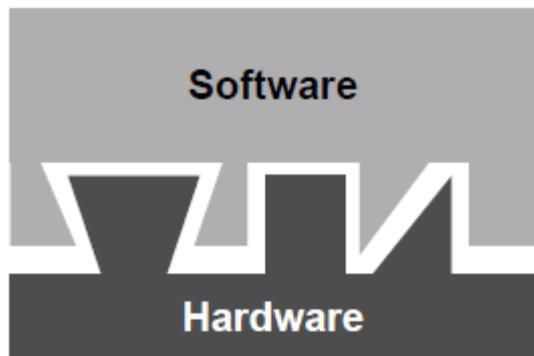
功能架构

物理架构



# 总结

提高复用性，缩短开发周期，提升开发质量，降低开发成本



# 参考文档

- 《AUTOSAR\_TechnicalOverview.pdf》
- 《AUTOSAR\_LayeredSoftwareArchitecture.pdf》
- 《AUTOSAR\_SWS\_VFB.pdf》
- 《AUTOSAR\_SWS\_RTE.pdf》
- 《AUTOSAR\_SoftwareComponentTemplate.pdf》

