**Autosar**

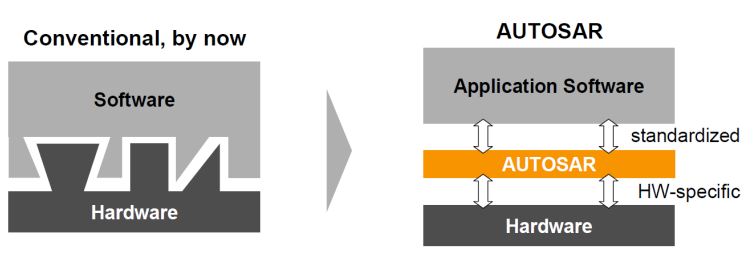
*Submitted by*

*Bhavin P V*

**Why AUTOSAR?**

The aim is not to optimize every single component, the future challenge for engineers is to optimize on a system level. In order to do so, a common platform is required that is scalable and supports exchangeable software modules. One of the fundamental ideas behind AUTOSAR is reusable Software Components that can deal with the increasing complexity today and in the future.

Software is tightly coupled with the ECU where it is going to be executed. If something is changed in the ECU, the software must be rewritten to suit with the hardware. It is problematic to buy software from one manufacturer and hardware from another, if they are not made to work with each other.

****

Benefits

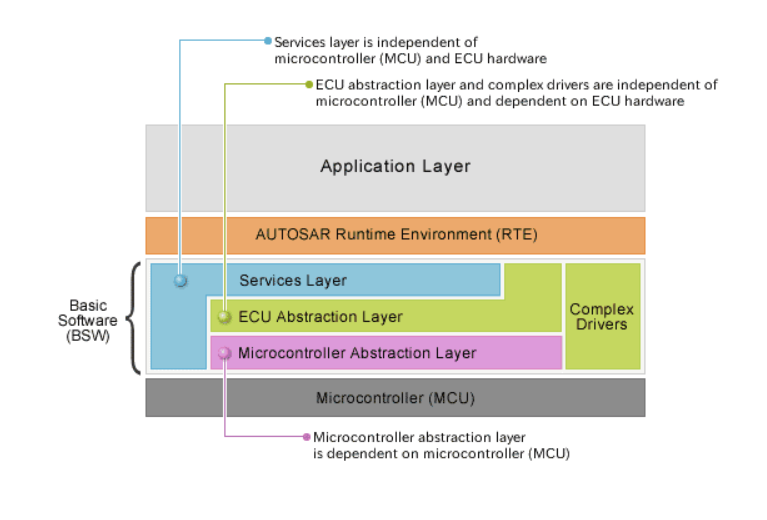
Modularity, Scalability, Transferability, Re-usability.

**AUTOSAR**

AUTOSAR (AUTomotive Open System ARchitecture) is a standardized software architecture framework for automotive systems. It provides a set of guidelines, specifications, and templates for developing automotive software that can be used across different electronic control units (ECUs) in a vehicle. AUTOSAR defines various structures to facilitate the development of software components and their integration within a distributed automotive system.

The AUTOSAR architecture is using a layered approach consisting of a total of three software layers running on top of a microcontroller. These three layers are called, starting at the bottom, BSW, RTE and finally Application Layer (AL).

**General AUTOSAR Structure**



**Application Layer**

The application layer is the first layer of the AUTOSAR software architecture and supports custom functionalities implementation. This layer consists of the specific software components and many applications which perform specific tasks as per instructions.

AUTOSAR ensures standardized interfaces for software components in the application layer and application software components help in generating simple applications to support the vehicle functions.

The communication between software components is enabled via specific ports using a virtual Function Bus. These ports also facilitate communication between software components and AUTOSAR Basic Software (BSW).

The AUTOSAR application layer consists mainly:

* + Application software components
  + Ports of software components
  + Port interfaces

**Runtime Environment**

RTE acts as middleware between the AUTOSAR application layer and lower layers. Basically, the RTE layer manages the inter- ECU and intra-ECU communication between application layer components as well as between BSW and application layer. The main task of the RTE is to make the layer above and below it completely independent of each other.

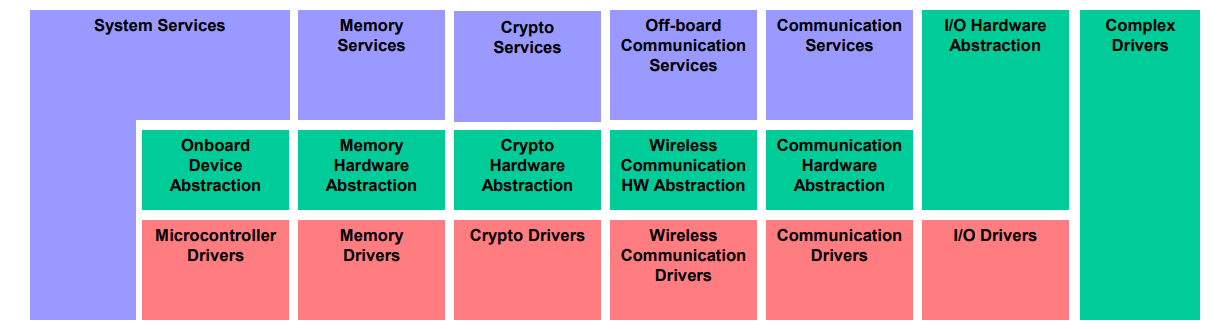
**Basic Software**

BSW can be defined as standardized software module offering various services necessary to run the functional part of upper software layer. This layer consists of ECU specific modules along with generic AUTOSAR modules.

Vertically the Basic Software can be grouped into:

* System
* Memory
* Communication
* Input/output (I/O)
* Complex Device Drivers

It is divided into three sub layers namely the Services layer, ECU (Electronic Control Unit) abstraction layer, and the Microcontroller Abstraction Layer (MCAL).



**BSW Layers**

#### **MCAL (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

**Implementation**: µC dependent

**Upper Interface**: standardized and µC independent

This layer consists of following module/blocks:

* **Microcontroller Drivers:** This module has drivers for accessing the internal peripherals of the MCU like Watchdog, General purpose Timers, or have functions for direct access to MCU like CoreTest.
* **Memory Drivers:** This module has drivers for accessing on chip memories like internal Flash, internal EEPROM, or memory mapped devices like external flash.
* **Crypto Drivers:** This module has drivers for on chip Crypto devices like SHE, HSM.
* **Wireless Communication Drivers:** This module has drivers for wireless network systems (in – vehicle or off board communication).
* **Communication Drivers:** This module has drivers for on board communication like [SPI](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface), [I2C](https://en.wikipedia.org/wiki/I%C2%B2C), etc and also has the drivers for Vehicle communication like [CAN](https://en.wikipedia.org/wiki/CAN_bus), [FlexRay](https://en.wikipedia.org/wiki/FlexRay).
* **I/O Drivers:** This module has drivers for accessing and using the I/O pins of MCU including digital and Analog, PWM.

#### **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

**Implementation**: µC independent, ECU hardware dependent

**Upper Interface**: µC and ECU hardware independent

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

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

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

**Properties**:

Implementation: µC independent, ECU hardware dependent

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

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

Example: An ECU has a microcontroller with 2 internal CAN channels and an additional on-board ASIC with 4 CAN controllers. The CAN-ASIC is connected to the microcontroller via SPI.

The communication drivers are accessed via bus specific interfaces (e.g. CAN Interface). **Task**: Provide equal mechanisms to access a bus channel regardless of its location (on-chip / on-board)

**Properties**:

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

Upper Interface: bus dependent, µC and ECU hardware independent.

* **Crypto Hardware Abstraction:** This module abstracts the crypto functionality by hiding the information of the Crypto used (internal, external device or software based). Again upper layers are unknown of the type of Crypto used or even which Crypto is used like whether it is on-chip or on-board or coded in software. This module provides mechanism for accessing the Crypto devices.
* **Memory Hardware Abstraction:** This module also abstracts the location of the memory device used. The memory can be on-chip or on-board and the on-board memory may have different ECU hardware layout (like the memory chip differences) but all this information is abstracted from application as it only knows about data and has no control over the type of memory device to be selected.
* **Onboard Device Abstraction:** This module provides abstraction from ECU specific on board devices. This module contains drivers for on board devices which cannot be called as sensors or actuators or timers like internal or external watchdog timers.

#### **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.

Services Layer offers: Operating system functionality, Vehicle network communication and management services, Memory services (NVRAM management), Diagnostic Services (including UDS communication, error memory and fault treatment), ECU state management, mode management, Logical and temporal program flow monitoring (Wdg manager)

**Task:** Provide basic services for applications, RTE and basic software modules.

**Properties**

Implementation: mostly µC and ECU hardware independent

Upper Interface: µC and ECU hardware independent

* **Communication Services:**  This is a group of modules for vehicle network communication which aims at providing uniform services for network management, providing uniform interface to vehicle network for communication and diagnostic communication, also to hide protocol and message properties from the application. Communication Services interfaces with the communication drivers (in MCAL) with the help of Communication Hardware Abstraction (as discussed above). This is MCU and ECU hardware independent but dependent on bus type. So, some part of these services may change if the bus type (CAN, FlexRay, etc) is changed.
* **Off board Communication Services:** This is a group of modules for vehicle to outer client’s communication via an adhoc wireless network. It has three blocks used for implementing different functionalities. This module provides a uniform interface for Wireless Ethernet network by hiding protocol and message properties.
* **Memory Services:** This service consists of one module, the NVRAM manager. It is responsible for the management of nonvolatile data (read/write from different memory drivers). Generally, application requires storing data in memory for future use, so this module is used to implement this in a uniform way and provide abstraction from memory locations and lower level properties. Memory Services provides mechanism for nonvolatile data management saving, loading, checksum, etc. This is MCU and ECU hardware independent and is highly configurable.
* **System Services:**This is a group of modules which can be used by modules of all layers. Examples are Real Time Operating System, Error messenger. These services can be some MCU dependent or may support special MCU capabilities (like Time Service), partly ECU Hardware and application dependent.
* **Crypto Services:** The Crypto Services consist of one module, the Crypto Service Manager. It is responsible for the management of cryptographic jobs and storage of keys.

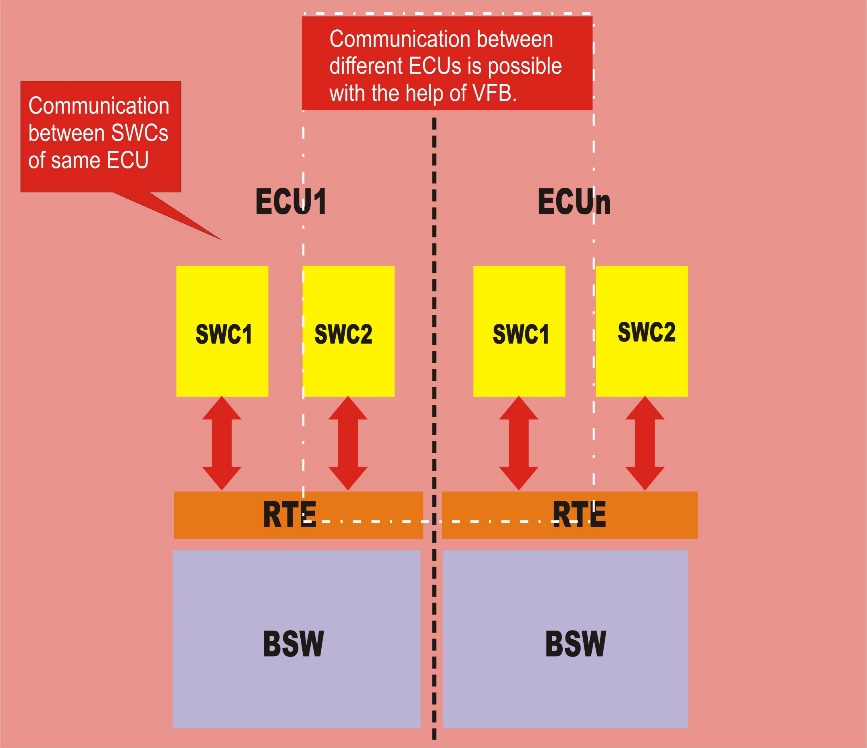
#### **Complex Device Drivers**:

This module useful in implementing nonstandard functionality within BSW stack. There can be many cases when we need to implement some functionality which is not supported by AUTOSAR, so CDD is used for such cases.  Or if a functionality requires strict timing constraints which may be lesser than the minimum timing of AUTOSAR OS resolution then Complex Device Drivers module is useful, as it directly helps connecting the MCU with the application.   
But complex driver’s functionality is highly MCU and application dependent and the code may not be easy to port.

**Virtual Function Bus**

The “**Virtual Function Bus**” symbolizes the communication between the Software components. The communication can be possible not only in **same ECU**but also **SWCs of other ECUs** in a system (vehicle). The communication mechanism to achieve this functionality is basically called as VFB, it is implemented in RTE layer of AUTOSAR.

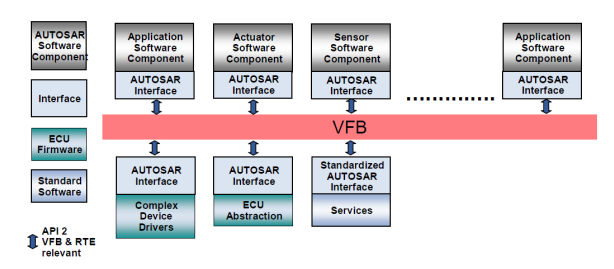
It’s called “virtual” because there is no physical connection between SWCs, instead the SWCs of a ECU communicate with SWCs of other ECUs in a system using low level communication bus like CAN or FlexRay. AUTOSAR handles all the low level matters to let the ECUs talk with each other as if there is a virtual link between them. Because of this SWC is independent of actual underlying hardware and hence relocate ability of SWCs is possible i.e. the same SWCs can be used on any ECU with any hardware platform.



As can be seen from above figure, the SWCs in same ECU can communicate with each other and also with SWCs in other ECUs of a system. Technically, communication between the SWCs of same ECU is called Intra-ECU Communication whereas the communication between SWCs of different ECUs is called Inter-ECU Communication.

Intra-ECU communication is realized virtually by defined function calls in RTE on the other hand the Inter-ECU communication is realized by using the underlying communication bus like CAN or Flex Ray through COM Stack along with RTE. If the ECUs use different communication bus, then a Gateway is required for translating data from one bus type to another bus type.

After all components has been chosen and defined, they are distributed to the relevant ECUs (mapping of software components to ECUs). The VFB is implemented with the help of an ECU-specific RTE (Run Time Environment). The RTE organize the communication between the individual components and with help of the operating system handles execution of the components. The RTE is scalable and generated statically for each ECU.



In contrast to the purely virtual specification of the communication topology and interaction between components which is done via the virtual function bus, the runtime environment provides an actual implementation for these artifacts. Although the concepts of virtual function bus and the runtime infrastructure are fundamentally different, they both share a common application programming interface. Since at the time the application modeler defines the interaction between its application and the virtual function bus, the application programming interface which is used at that time has to be used by the runtime environment as well in order to provide a working environment for the modeled application. However, in other terms like system view or concepts of communication supported, the VFB and the RTE have a fundamentally different conceptual base.

# RTE

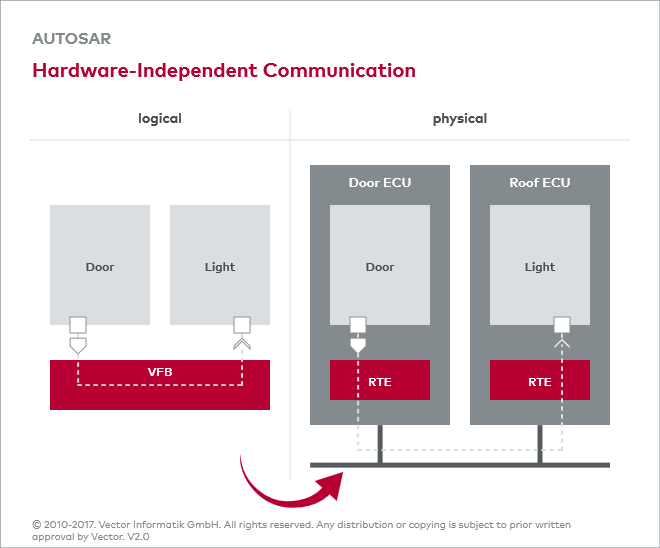
The RTE is the layer between the application layer and the BSW layer. It provides the application software with services from the service layer. All communication between SWCs, either on the same ECU or different ones, or services are done via the RTE. The main task of the RTE is to make the layer above and below it completely independent of each other. In other words, SWCs running on an ECU have no idea what the ECU looks like hence a SWC will be able to run on different looking ECUs without any modifications.

Logically the RTE can be seen as two sub-parts realizing different SWC functionality

1)Communication

2) Scheduling

When the RTE is implemented it is ECU and application dependent; thus it is specifically generated for each different looking ECU. Instead of adapting SWCs to different ECUs that is what is done with the RTE, this way SWCs can stay the same thus accomplishing the set out task.

****

The RTE along with AUTOSAR COM, OS and other BSW modules is the implementation of VFB Concept for a ECU. All the ports and interfaces are implemented in RTE which thereby realize the communication between SWCs and also act as a means by which SWC can access BSW modules like OS and Communication services. As described above, RTE has interfaces using which the Runnable in SWC communicates with other SWC or BSW module. RTE maps the Runnable to OS tasks as configured during RTE configuration and executes runnable either in same task or differently. RTE also handles the work of triggering runnable (if conditions are met) using RTE events as configured in SWC configuration. RTE is strongly linked with BSW scheduler due to same OS task may be used for both scheduling of SWCs and schedulable entities (also called main processing functions) of BSW module.

RTE is generated after integration of SWC, so RTE is responsible to make sure that system is working as expected by ensuring the communication of SWCs (between themselves as well as with BSW modules) is smooth irrespective of where the SWCs are deployed. RTE supports both SWCs with source code as well as SWCs with only object codes. RTE don’t support any runtime reconfiguration, i.e. every communication between SWCs and BSW modules must be configured before RTE is generated.

**Other uses of RTE:**

* Invoke and support multiple Runnables of SWC based on different RTE events as per configuration.
* Include any SWCs from any project during configuration and perform the operation as expected to accomplish the SWC reuse ability and portability feature of AUTOSAR.
* As RTE handles the execution of runnable in OS task, it implements **activation offset** which is used to optimize CPU load, memory etc, in case of time triggered runnable mapped in same OS task. Because time triggered variables need to be executed on configured time and avoid conflicting of runnables by overlapping, RTE uses activation offset. RTE handles this by calculating the **maximum period of task** which is the Greatest Common Divisor(GCD) of all runnables and it assumes that any runnable maximum execution period would be lower than GCD.
* RTE also notifies runnables of any interrupt (if configured) occurred at lower layers, but this doesn’t mean runnables will be executed in ISR. Runnables are subset of SWC and SWC is completely independent of lower layers.
* RTE has to ensure data consistency when sharing variables among runnables of same SWC, or inter partition SWC or intra partition SWC or even in communication between SWC of different ECUs.
* RTE takes care to prevent any conflict if senders transmit at same time to one receiver or vice versa.

# Application Software Component

## Introduction

The application layer consists of the specific software components and many applications which perform specific tasks as per instructions.

The AUTOSAR standard specifies the application layer implementation using a “component” concept.

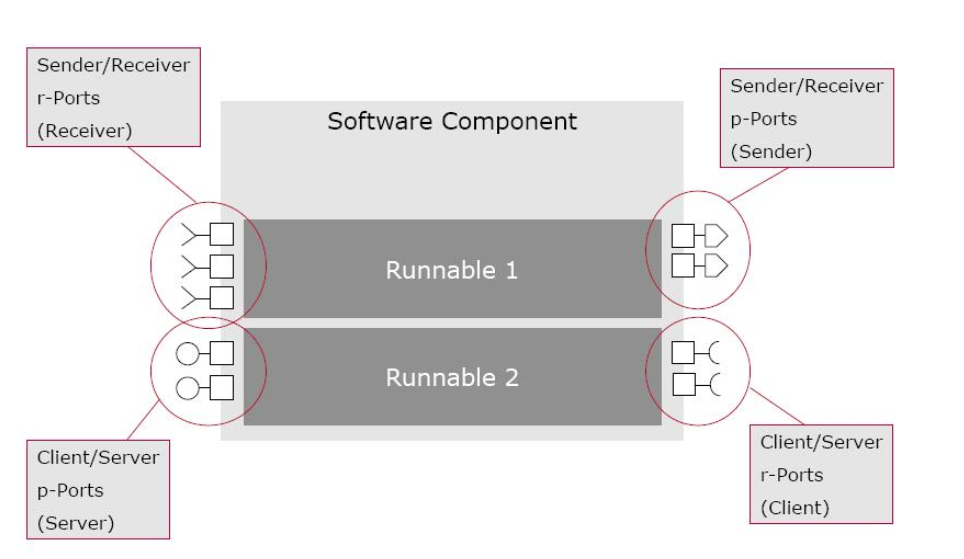
The AUTOSAR application layer consists of three components which are:

Application software components

Ports of software components

Port interfaces.

Software Component is not only logic block, it also contains ports, port interfaces, runnable entities, internal behavior, configuration parameters.



## 

**SWC description includes**

* General characteristics

1. Name 2) Manufacturer

* Communication properties

1. P ports 2) R ports 3) Interfaces

* Inner structure (composition)

1. Sub-components 2) Connections 3) Runnable

* Required hardware resources

1. Processing time 2) Scheduling 3) Memory

SWC s interact with each other as well as with other modules (BSW) in the lower layer through its ports and port interfaces.

SWCs to communicate with each other they use the virtual functional bus (VFB). From a SWC’s point of view all it sees is the VFB and not the hardware dependent BSW and the hardware itself; in reality this is provided by the RTE.

## Main Types of Software Components

* + Atomic Component
    - It cannot be further subdivided and can be assigned to only one ECU.

It implements a part of the application.

* + - * + Application: Implement the algorithm
        + Sensor/Actuator Component: It handles sensor evaluation and actuator control functions. It depends on associated sensor/actuator and independent of ECU mapped to it.
  + Composite Component
    - It offers a logical interconnection of other components, which could be either atomic or composite. It can be deployed in different ECUs
  + Calibration parameter component
* To provide values for calibrating all connected components. If all connected components are not located on the same ECU, the calibration parameter component has to be duplicated and placed on every ECU having at least one other component needing it. A calibration parameter component does not have any internal behavior as oppose to “normal” SWC.
  + Service component
* A service component in AUTOSAR is standardized and uses standardized interfaces to provide services. Direct access to BSW is needed in order to provide these services to other components.
  + ECU-abstraction component
* When the sensor-actuator component was described above it was mentioned that it interacted with the ECUAL. To be more precise, it provided access to I/O via a client-server P Port on an ECU abstraction component. This ECU-abstraction component has the ability to interact with other BSW.
  + Complex device driver component
* It is a generalization of the ECU-abstraction component. It can communicate directly with other BSW and is typically used for applications with critical resource demands.

**Benefits of SWCs:**

* + - Modularity
    - Portability
    - Scalability
    - Interoperability

## Ports

A port belongs to SWC and it act as the interaction point between a two SWC. A port is associated with a port-interface.

Contents: Data elements (Sender/ Receiver) and Operations (Client / Server).

**Types**

**Provided Ports (P port):** Provided Ports are connection points within a SWC that allow it to expose its services, data, or functionality to other SWCs. They represent the SWC's outputs or provided interfaces. Other SWCs can connect to the Provided Ports to access the services or data provided by the SWC.

**Required Ports (R port)**: Required Ports are connection points within a SWC that enable it to receive services, data, or functionality from other SWCs. They represent the SWC's inputs or required interfaces. The SWC connects to the Required Ports to access the services or data it depends on from other SWCs.

By connecting the P Port of one SWC to the Required Port of another SWC, AUTOSAR ensures a standardized and well-defined communication mechanism between SWCs within the system.

## Ports Interfaces

An interface is attached to a port and it describes the data or operations that are provided or required by a SWC via that port.

**Types**

* + Sender Receiver Port Interface
  + Client Server Port Interface

**Client Server Port Interface**

The server provides services while one or more clients may use its services to carry out tasks.

Contents: Operations

Communication: 1:1 or n:1 (Multiple client to single server)

Mainly used in diagnostics.

An operation invoked on a server uses arguments supplied by a client to perform the requested task, these arguments can either be of simple (bool) or more complex nature (array). When a client wants to invoke an operation of a server’s interface, a value for each operator parameter has to be provided. At a later point in time when a response is received it can either be a good or a bad one.

There are a total of three possible responses:

**Valid response**: Server was able to execute the requested operation and values have been assigned to all parameters provided by the operation interface.

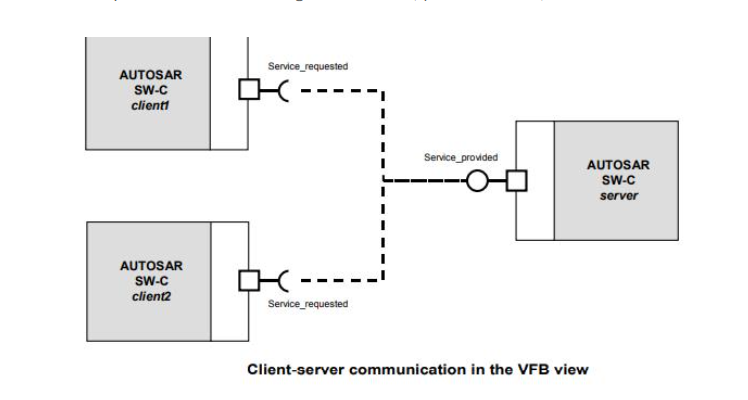
**Infrastructural-error**: A transmission to/from the server went wrong due to a broken bus.

**Application-error:** Something went wrong on the server-side while executing the operation invoked by a client.

A component acting as a client has to have an R Port connected to exactly one of a server’s P Ports. This P Port can however be connected to an arbitrary number of client R Ports.

Since a component is not limited to one port, one and the same component can act as both a client and a server. Limits exist on how clients may invoke operations on servers. Two concurrent invocations of the same operation on the same client R Port is not allowed; not until a response on the first invocation has been received, a good or a bad one.

Concurrent invocations of different operations are allowed on the same R Port but the VFB does not guarantee the ordering of the invocations. The VFB is however required to enable a client to relate a response to a previous invocation.

****

AUTOSAR SWCs act as client 1 and 2 respectively, request a service through the R Port which is sent to the server AUTOSAR SWC which offers the required services through the P Port.

**Sender Receiver Port Interface**

This way of communicating allows one receiver to get information from multiple senders or a sender to send information to an arbitrary number of receivers.

It defines an asynchronous information distribution and allows for a more data-oriented information exchange over the virtual function bus.

Contents: Data Elements

Communication: 1: n or n:1

There are two ways for which data-elements can be provided by the sender:

Last-is-best: Last value made available by the sender is the current one.

Queued: Values are stored in a queue of predefined size.

A sender is completely decoupled from any receivers; it has no idea how many (if any) receivers are using the values it is producing. It is however possible at configuration time to enable transmission acknowledgements if a sender wants to know if a transmission succeeded or not.

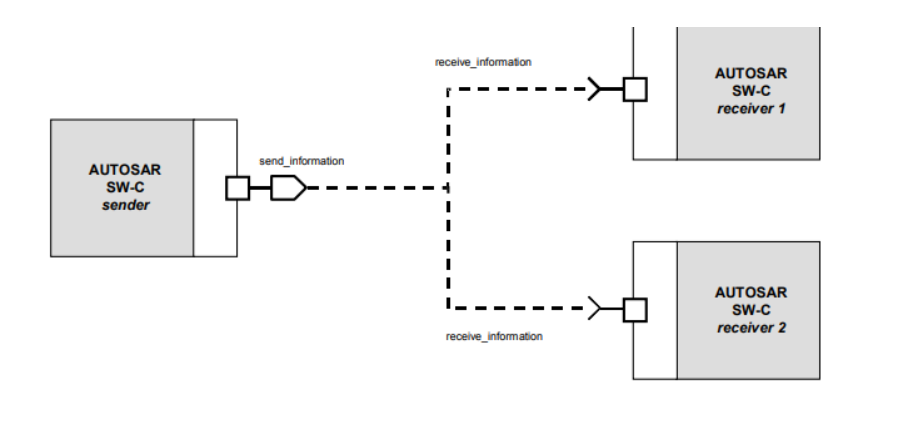
If the “queued” semantics is used the receiver consumes values from the queue in a FIFO-fashion. The queue is located on the receiver side of the connection; this is logical since consumable reads are taking place.

If a queue is full and there are still new values to add a queue overflow will occur and the new values are discarded; whether or not the receiver is notified of these events can be set at configuration time.

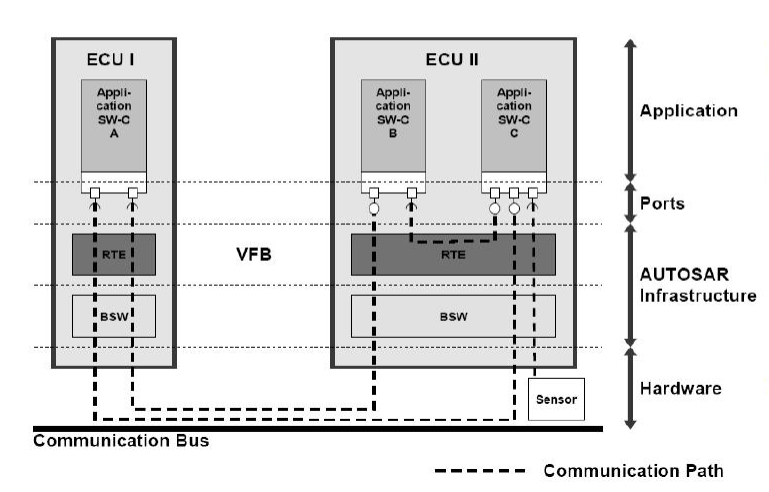
For “Last-is-best”, only 1: n (1 sender, n ≥ 0 receivers) communication is possible.

For “Queued” semantics both 1: n and n:1 (n ≥ 0 senders, 1 receiver) are possible.

The latter is possible because multiple senders can add values to the queue of a single receiver. For “last-is-best” there is only one value and if there are multiple senders they would constantly overwrite each other’s values and there is no guarantee the receiver will be able to see all produced values.



**Inter and Intra ECU Communication Diagram**

****

## AUTOSAR Data Types

**ApplicationDatatype**Define all data attributes that are needed from the application point of view. It is used for exchanging data between SWC or between a SWC and a measurement and calibration tool. For this level also the communication for a complete VFB can be specified.

Application Data Level

* Application Data Types and their Data Prototypes for Application SWC types are defined from the application point of view.
* For integration of the SWC into the ECU, the Application Data Types must be mapped to Implementation Data Types.
* An Application Data Type Level describes the purpose and limitation of a value that can be variable or information sent & received between multiple components.
* Application Data Type further derived into
* Application Primitive Data Type
* Application Composite Data Type:

Application Record Data Type: Equivalent to a strut {} in C

Application Array Data Type: Equivalent to an array [] in C

Application Primitive Data Type

An Application Primitive Data Type describes how to transform an internal application value to the corresponding value in the physical domain or vice versa. It does not, however say anything about the implementation choices, such as final data types.

* Computation Method: The Computation Method describes the translation of Application internal values to the corresponding values in physical domain for specific interval and vice versa
  + SW Unit: SW Unit describes about a physical unit.
* Data Constraint: Data Constraint describes which range that this specific Application Primitive Data Type is allowed to operate in. The purpose is to define the allowed operational interval for a certain data type.

Application Composite Data Type

* Structures and data arrays can be defined on Application data level via Application Composite Data Types. The corresponding types are Application Record Data Types and Application Array Data Types.

Application Record Data Type: This Data Type describes a complex data type. A complex data type is used to group other data types that have something in common.

Application Array Data Type: This Data Type describes an array of objects. The number of objects in the array is specified by defining the attribute value for Max no. of elements. Whether the array can vary in size or not, is defined by the attribute Array Size Semantics.

**Implementation Data Type**

Enable a formal definition of types from the implementation point of view. They Are used for interface descriptions and data within the basic software. The data description in this level is also input for the RTE generator.

These data types are not exposed to the application layer or the external interfaces but are used for internal operations and computations within the software components. They used within the lower layers such as the RTE and BSW modules.

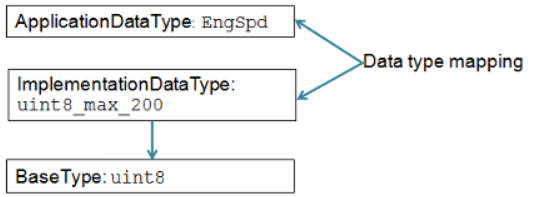
These types conceptually represent the source of code of a programming language like C

**Base Data Type**

Describe the primitive elements in terms of bits and bytes. They are also the base for the implementation data type. The attributes for base data type can be defined either platform independent or platform specific. These base data types provide the basic building blocks for defining more complex data structures, such as arrays, structures, enumerations, or pointers. The description of base types is required for generation of RTE.

AUTOSAR standard defines an approach to Autosar data types in which base data types are mapped to implementation and application data types.

Application and implementation data types separate application-level physical attributes, such as real-world range of values, data structure, and physical semantics, from implementation-level attributes, such as stored-integer minimum and maximum and specification of a primitive type (for example, integer, Boolean, or real).



**Runnable**

A runnable is the smallest code fragment inside a SWC. It is these runnable that are mapped to OS tasks and will execute the behavior of the SWC. This is needed since the SWCs themselves are not aware of the BSW layer where all the OS functionality resides. A SWC can have multiple runnable which may be required to carry out certain tasks.

Each runnable is implemented by exactly one C-function. They are the smallest units provided by the Software Component which can be triggered and executed by the RTE by a

* Timing event which triggers the runnable entity periodically
* Data received event which triggers the runnable entity as soon as a variable data prototype is received at the port.

A Runnable Entity can be executed and scheduled independently from the other Runnable Entities. It is described by a sequence of instructions that can be started by the RTE. Each runnable entity is associated with exactly one Entry Point. A Runnable Entity contains at least two points for the Scheduler: One Entry Point and one Exit Point.

Types of Runnables in AUTOSAR

In AUTOSAR, there are two main types of Runnable Entity as per the Operating System:

* **Background Runnable Entity**: This type of Runnable Entity In AUTOSAR is executed in is used to perform tasks that are not time-critical or that do not require immediate attention.
* **Foreground Runnable Entity**: This type of Runnable Entity is executed in the foreground, typically at a higher priority than background Runnable Entities. It is used to perform tasks that are time-critical or that require immediate attention, such as tasks that involve safety-critical functions or real-time control.

There are 3 types of runnable in AUTOSAR architecture.

* **Init Runnable**: In AUTOSAR, an Init Runnable is a software routine that is executed during the initialization phase of an ECU. It is used to perform tasks such as setting up hardware and software resources, configuring communication channels, and preparing the ECU for normal operation. Init Runnable may also be triggered by events or conditions, such as the detection of a fault or the receipt of a command from another component.
* **Periodic Runnable**: If the runnable is configured as periodic, then it will be getting called periodically as per the configured time. This is handled by the OSEK scheduler. This is also called sender or receiver type runnable. It is typically used to perform tasks that need to be executed on a periodic basis, such as reading data from sensors or updating the state of a system.
* **Server Runnable**: In AUTOSAR, a Server Runnable is a type of Runnable Entity that is used to provide a service to other components in the system. It is typically triggered by an external request, such as a request from another component to access a specific piece of data or perform a specific task. This runnable is used to handle the functions event based. This is nothing but the client-server communication type.

## 