| **Document Title** | # Specification of the Virtual Functional Bus |
|---|---|
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 056 |
| **Document Classification** | Auxiliary |

| | |
|---|---|
| **Document Version** | 1.3.0 |
| **Document Status** | Final |
| **Part of Release** | 3.2 |
| **Revision** | 2 |

**Document Change History**

| Date | Version | Changed by | Change Description |
|---|---|---|---|
| 17.05.2012 | 1.3.0 | AUTOSAR Administration | • Support of NV data communication at element level |
| 18.03.2011 | 1.2.0 | AUTOSAR Administration | • Legal Disclaimer Revised<br>• Added description of categories of runnables |
| 14.07.2010 | 1.1.0 | AUTOSAR Administration | Last-is-best N:1 S/R communication allowed |
| 23.06.2008 | 1.0.1 | AUTOSAR Administration | Legal Disclaimer Revised |
| 14.11.2007 | 1.0.0 | AUTOSAR Administration | Initial Release |

**Advice for users**

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

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

# Table of Content

# 1 Introduction to this document

## 1.1 Contents

This specification describes the AUTOSAR Virtual Functional Bus (VFB).

## 1.2 Prereads

This document is one of the high-level conceptual documents of AUTOSAR.
The only required pre-read is [AUTOSAR Technical Overview]. Useful pre-reads are [Main Requirements] and [AUTOSAR Methodology]. Documents that can be consulted in parallel to this document include the glossary [AUTOSAR Glossary].

## 1.3 Relationship to other AUTOSAR specifications

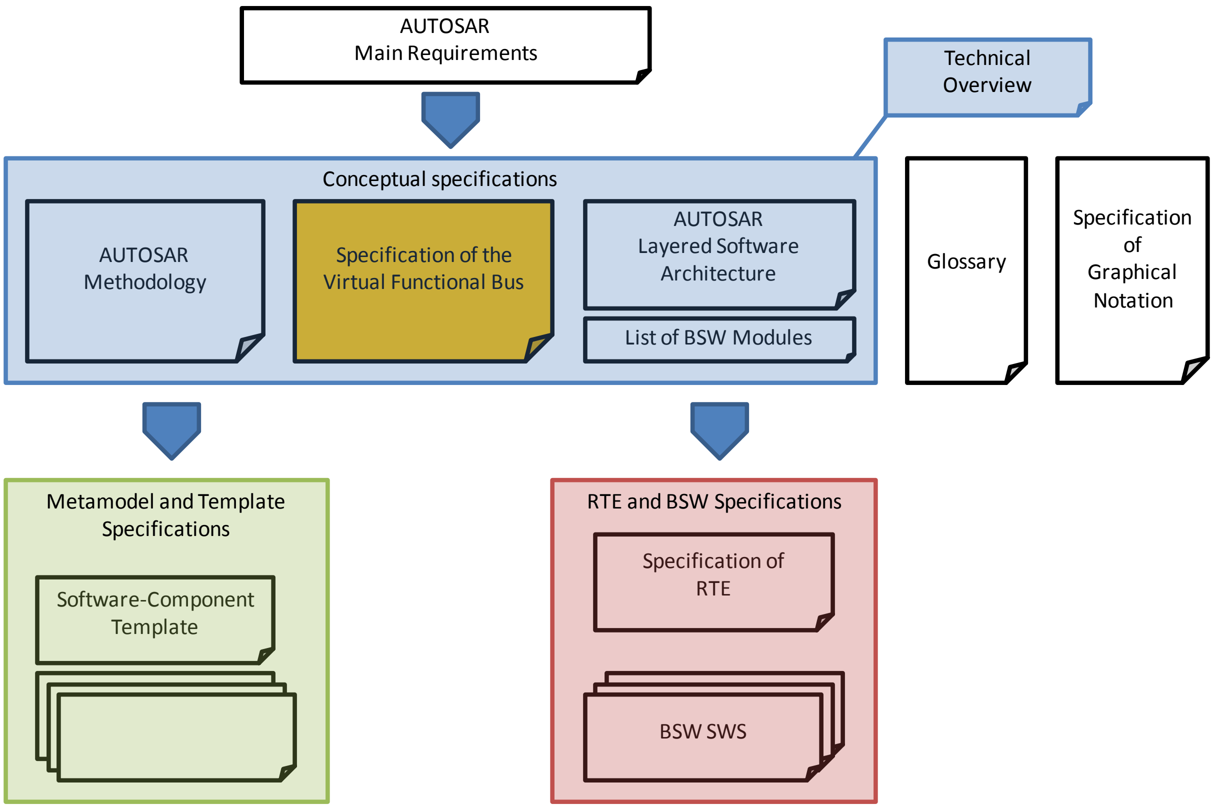


**Figure 1.1: Relationship of the "Specification of the Virtual Functional Bus" to other specifications**

Figure 1.1 illustrates the relationship between the "Specification of the Virtual Functional Bus" and other major AUTOSAR specifications. The "Specification of the Virtual Functional Bus" is part of a set of specifications describing the overall concepts of AUTOSAR. These documents give a conceptual overview of AUTOSAR and serve as requirements to the more detailed specifications. The conceptual specifications include:

- The "AUTOSAR Methodology" [AUTOSAR Methodology] describes the method that is used when building systems with AUTOSAR
- The "Specification of the Virtual Functional Bus"
- The "Layered Software Architecture" [Layered Software Architecture]

These conceptual documents are refined and made concrete into a large set of AUTOSAR specifications, which can be grouped into:

- the specifications defining the AUTOSAR meta-model and templates; in this group the "Software-Component Template" [Software Component Template] is directly influenced by the VFB concepts
- the specifications defining the AUTOSAR basic-software modules and the RTE; in this group the "Specification of RTE" [Specification of RTE Software] is directly influenced by the VFB concepts.

## 1.4 Structure and conventions of this document

### 1.4.1 Structure of this document

Figure 1.2 shows the structure of this document. The first chapters define the VFB concepts generically and should be read in order. The last chapters define and clarify specific issues, such as the interaction with hardware, mode-management, AUTOSAR-Services or Measurement and Calibration. The chapter about the timing model is for information purposes only and is not part of the standard. It is made available to show the early conceptual work to model time aspects in the VFB.

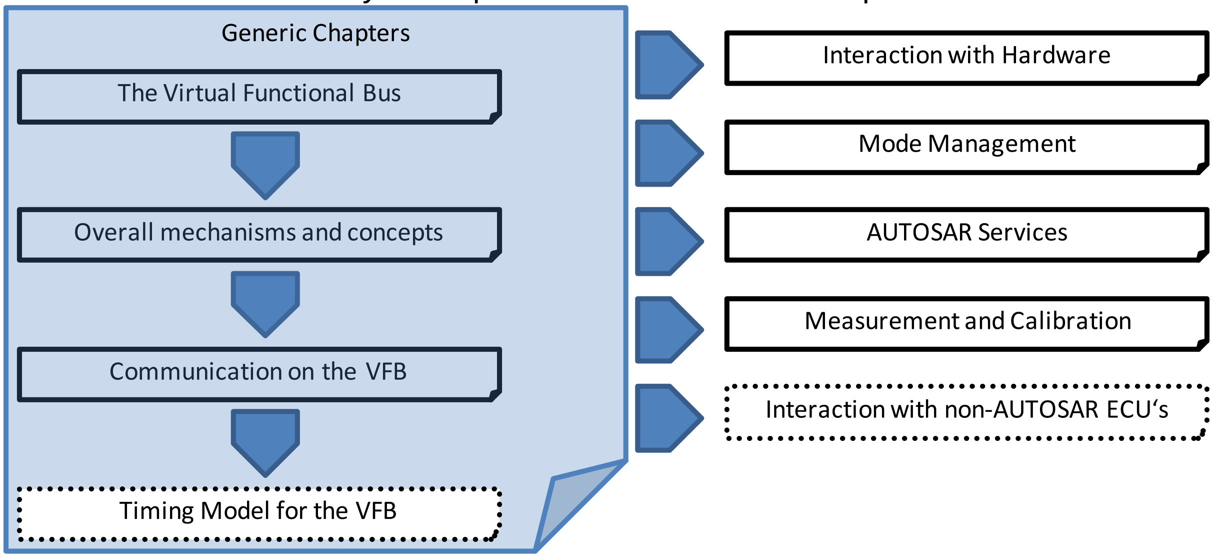

Generic Chapters
The Virtual Functional Bus
Overall mechanisms and concepts
Communication on the VFB
Timing Model for the VFB
Interaction with Hardware
Mode Management
AUTOSAR Services
Measurement and Calibration
Interaction with non-AUTOSAR ECU's

**Figure 1.2: Structure of the document**

### 1.4.2 Specification Items

The requirements on the “Virtual Functional Bus” resulting from this document are listed explicitly as numbered “specification items”. Each specification item has a unique ID of the form “VFB-XXX” and has the following format:

| |
|---|
| VBF-XXX : Example of a specification Item |

# 2 The Virtual Functional Bus

Figure 2.1 shows an overview out of the [AUTOSAR Methodology]. Figure 2.2 illustrates the “Configure System” activity out of the methodology (top-left), focusing on the VFB.

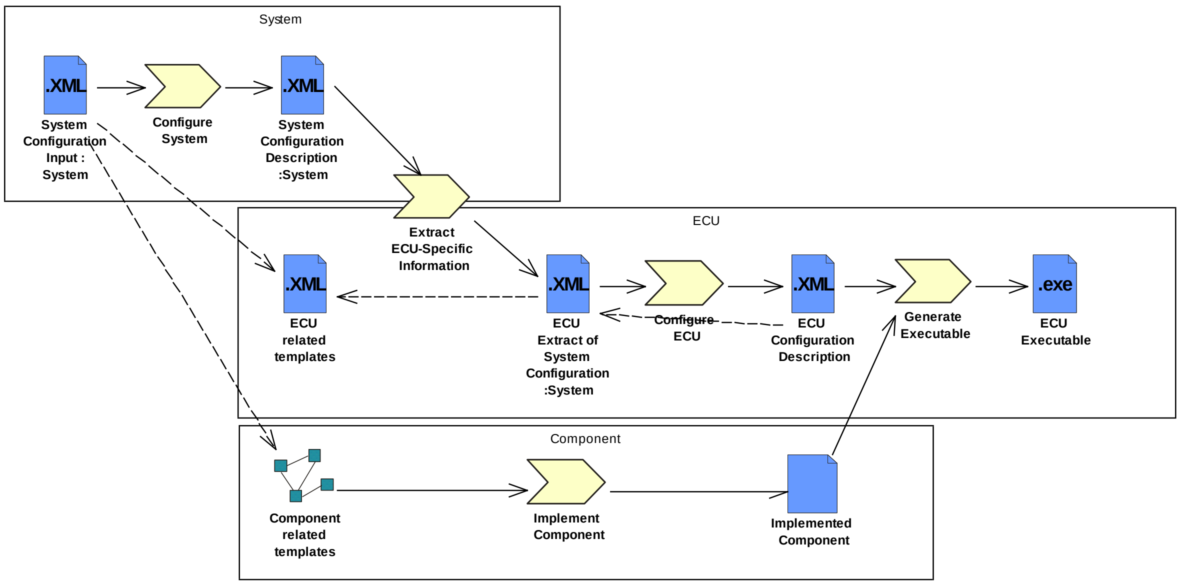

System
.XML
System Configuration Input : System
Configure System
.XML
System Configuration Description :System
Extract ECU-Specific Information
ECU
.XML
ECU related templates
.XML
ECU Extract of System Configuration :System
Configure ECU
.XML
ECU Configuration Description
Generate Executable
.exe
ECU Executable
Component
Component related templates
Implement Component
Implemented Component

**Figure 2.1: Methodology Overview out of [AUTOSAR Methodology]**

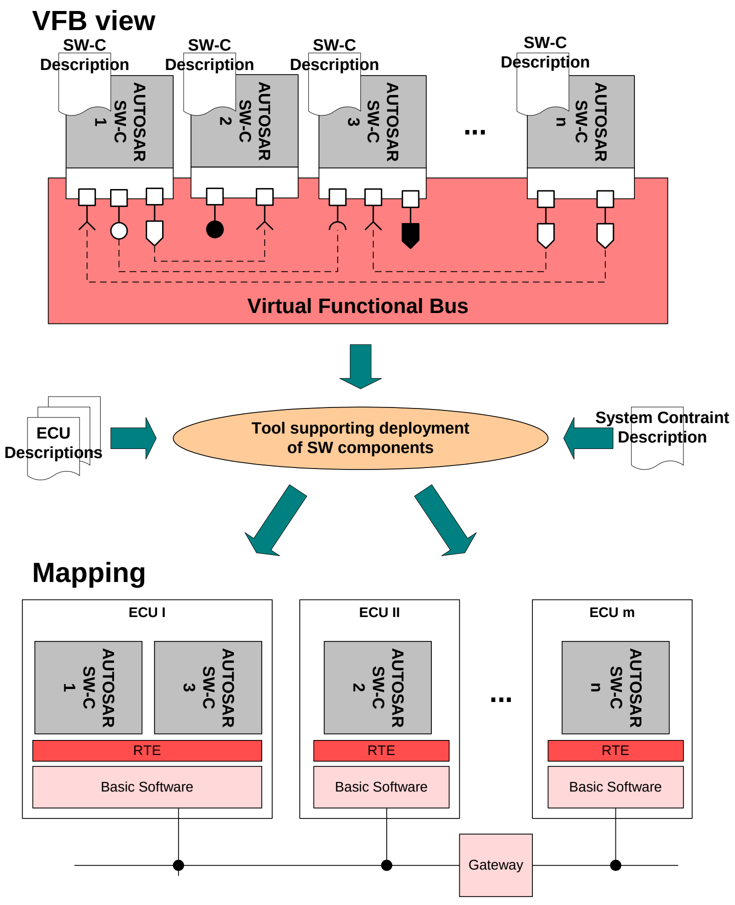


**Figure 2.2: Detailed view on the activity "Configure System"**

In AUTOSAR, an application is modeled as a composition of interconnected components. This is illustrated in the top half of Figure 2.2 (labeled "VFB view"). The "virtual functional bus" is the communication mechanism that allows these components to interact. In a design step called "Configure System", the components are mapped on specific system resources (ECUs). Thereby, the virtual connections between the components are mapped onto local connections (within a single ECU) or on network-technology specific communication mechanisms (such as CAN or FlexRay frames). Finally, the individual ECUs in such a system can be configured. The concrete interface between the components and the rest of the system on an ECU is called the Run-Time Environment (RTE), which is defined in [Specification of RTE Software].
A component encapsulates complete or partial automotive functionality. Components consist of an implementation and of an associated formal software-component description (defined in [Software Component Template]). The concept of the virtual functional bus allows for a strict separation between applications and infrastructure. The software components implementing the application are largely independent of the communication mechanisms through which the component interacts with other components or with hardware (such as sensor or actuators). This fulfills AUTOSAR's goal of relocatability (also see [Main Requirements]).
With this the complete communication of a system can be specified including all communication sources and sinks. The VFB can therefore be used for plausibility checks concerning the communication of software components. The communication

connections and the connected software components are saved in one description, which will be used for the next process steps (mapping, software configuration, etc.).
The VFB specification needs to provide concepts for all infrastructure-services that are needed by a component implementing an automotive application. These include:

- Communication to other components in the system
- Communication to sensors and actuators in the system (see Chapter 6, Interaction with hardware)
- Access to standardized services, such as reading to or writing from non-volatile ram (see Chapter 7, AUTOSAR Services)
- Responding to mode-changes, such as changes in the power-status of the local ECU (see Chapter 8, Mode Management)
- Interacting with calibration and measurement systems (see Chapter 9)

# 3 Overall mechanisms and concepts

## 3.1 Components

The central structural element used when building a system at the VFB-level is the "component". A component has well-defined "ports", through which the component can interact with other components. A port always belongs to exactly one component and represents a point of interaction between a component and other components.
Figure 3.1 shows an example of the definition of a component-type called "SeatHeatingControl", which controls the heating element in a seat based on several information sources. In this example, the component inputs whether a passenger is sitting on the seat (through the port "SeatSwitch"), the setting of the seat temperature dial (through the port "Setting") and some information from a central power management system (through the port "PowerManagement"), which could decide to disable seat heating in certain conditions. The component controls an LED associated with the seat temperature dial (port "DialLED") and the heating element (through the port "HeatingElement"). Finally, the component can be calibrated (port "Calibration"), needs the status of the ECU on which the component runs (port "ecuMode") and requires access to local non-volatile memory (port "nv").

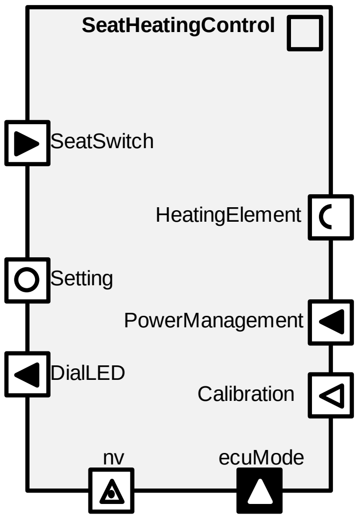


**Figure 3.1: Example of the definition of the component-type "SeatHeatingControl" with eight ports**

Figure 3.2 shows an example of the definition of a sensor-actuator component[1] called "SeatHeating". This component inputs the desired setting of the heating element (through the port "Setting") and directly controls the seat heating hardware (through the port "IO").

[1] Chapter 6, Interaction with hardware, defines the exact purpose of the "sensor-actuator" components

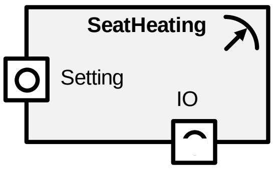


**Figure 3.2: Example of the definition of a component-type "SeatHeating" with two ports**

A single component can implement both very simple but also very complex functionality. A component may have a small number of ports providing or requiring simple pieces of information, but can also have a large number of ports providing or requiring complex combinations of data and operations.
AUTOSAR supports multiple instantiation of components. This means that there can be several instances[2] of the same component in a vehicle system. Figure 3.3 shows how two instances of the "SeatHeatingControl" component-type are used to control the left front seat, respectively the right front seat. These components will typically have their own separate internal state (stored in separate memory locations) but might for example share the same code (in as far as the code is appropriately written to support this).

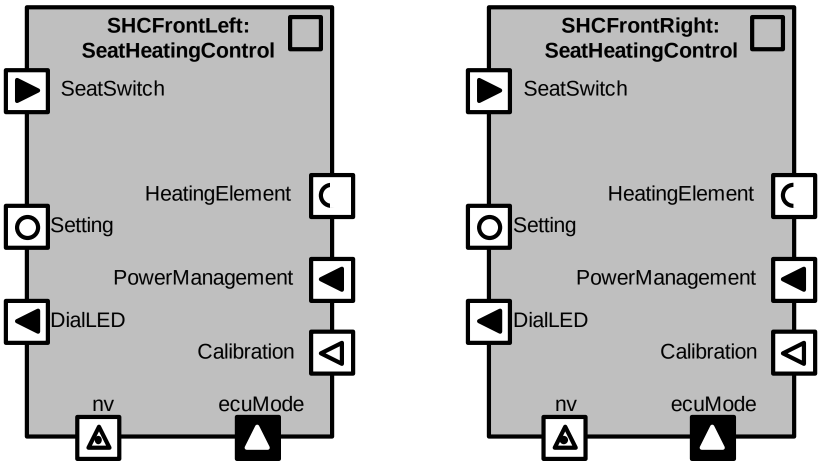


**Figure 3.3: Example showing the multiple instantiation of the component "SeatHeatingControl" as "SHCFrontLeft" and "SHCFrontRight"**

VFB001: At configuration time, the component's ports are known

VFB002: Components interact with each other through their ports only

VFB084: A component-type can be instantiated multiple times on the VFB

[2] Dynamic instantiation at runtime is not scope of the present release of AUTOSAR.

## 3.2 Port-Interfaces

A port of a component is associated with a “port-interface”. The port-interface defines the contract that must be fulfilled by the port providing or requiring that interface.

VFB003: At configuration time, each port is typed by exactly one port-interface

Table 3.1 lists the three kinds of port-interfaces supported by AUTOSAR: client-server, sender-receiver and calibration.

| **Kind of port-interface** | **Comment** | **Further reading** |
|---|---|---|
| Client-server | The server is provider of operations and several clients can invoke those operations. | this section and Section 4.4 |
| Sender-receiver | A sender distributes information to one or several receivers, or one receiver gets information (events) from several senders[3]. A mode manager can notify mode switches to one or several receivers | this section and Section 4.3 |
| Non volatile Data Interface | Provide element level access (read only or read/write) to non volatile data as opposed to NV block access. | |
| Calibration | Calibration is a static communication pattern: it allows modules to access static calibration parameters. | Chapter 9 |

**Table 3.1: The kinds of port-interfaces provided by AUTOSAR.**

A client-server interface defines a set of operations that can be invoked by a client and implemented by a server. Figure 3.4 shows an example of the definition of a simple client-server interface. The interface “HeatingElementControl” defines a single operation called “SetPower” with a single ingoing argument called “Power”. The operation can return an application error called “HardwareProblem”.

[3] In the context of AUTOSAR, sending, receiving and distributing of events is seen as part of the sender-receiver communication pattern.

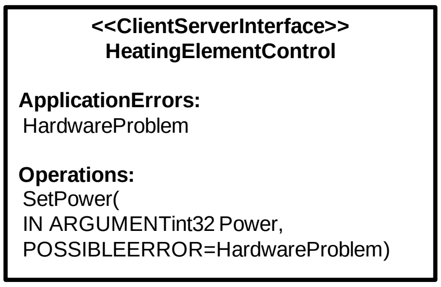


**Figure 3.4: Example of a client-server interface “HeatingElementControl” with a single operation**

A sender-receiver interface defines a set of data-elements that are sent and received over the VFB. Figure 3.5 shows the definition of a simple sender-receiver interface called “SeatSwitch” containing a single data-element called “PassengerDetected”.

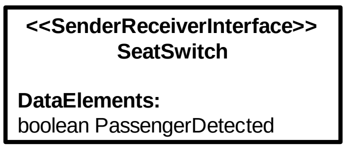


**Figure 3.5: Example of a Sender-Receiver Interface “SeatSwitch” with a single data-element**

VFB004: At configuration time it is known whether the port-interface is a client-server interface or a sender-receiver interface

VFB005: At configuration time, it is known which operations a client-server interface contains

VFB006: At configuration time, it is known which data-elements a sender-receiver interface contains

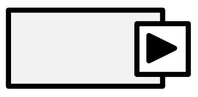
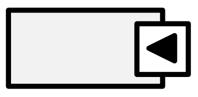
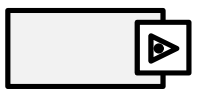
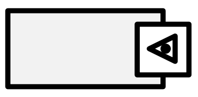
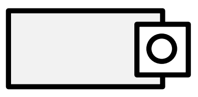
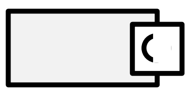
## 3.3 Ports

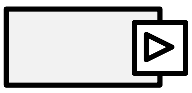
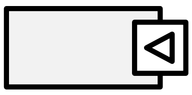
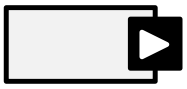
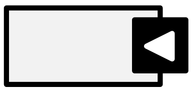
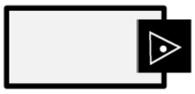
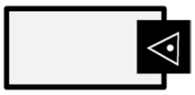
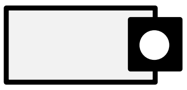
As defined before, the ports of a component are the interaction points between components.
A port of a component is either a “PPort” or an “RPort”. A “PPort” provides the elements defined in a port-interface. An “RPort” requires the elements defined in a port-interface. A port is thus typed by exactly one port-interface[4]. A single port-interface can type several different ports.

VFB007: At configuration time, it is known whether a component’s port is a PPort or an RPort

[4] This implies that a port only provides one elementary communication pattern (either sender-receiver or client-server). This is necessary because otherwise a reasonable connection of ports is not possible. Additionally only in this way a reasonable modeling e.g. of data flow is possible.

Table 3.2 shows the port-icons for the various combinations and summarizes the semantics of those ports.

| Kind of Port | Kind of Interface | Service Port | Port-Icon and description |
|---|---|---|---|
| PPort | sender-receiver | No |  The component provides values for the data-elements and mode-groups in the interface |
| RPort | sender-receiver | No |  The component reads or consumes values for the data-elements and mode-groups in the interface |
| PPort | NV data | No |  The NV Block Component provides access to non volatile data |
| RPort | NV data | No |  The component requires access to non volatile data provided by an NV Block Component |
| PPort | client-server | No |  The component provides (=implements) the operations defined in the interface |
| RPort | client-server | No |  The component requires (=uses or invokes) the operations defined in the interfaces |

| | | | |
|---|---|---|---|
| PPort | calibration | No |  The component provides calibration data |
| RPort | calibration | No |  The component requires calibration data |
| PPort | sender-receiver | Yes |  The component provides data-elements and mode-groups to an AUTOSAR Service |
| RPort | sender-receiver | Yes |  The component reads/consumes data-elements and mode-groups from an AUTOSAR Service |
| PPort | NV data | Yes |  The component provides access to non volatile data to an AUTOSAR service |
| RPort | NV data | Yes |  The component requires access to non volatile data provided by an AUTOSAR service |
| PPort | client-server | Yes |  The component provides (=implements) operations for an AUTOSAR Service |

| | | | |
|---|---|---|---|
| RPort | client-server | Yes | The component invokes operations from an AUTOSAR Service |

**Table 3.2: Semantics of the port-icons**

When a PPort of a component provides a client-server interface, the component to which the port belongs provides an implementation of the operations defined in the interface.
In the example of Figure 3.6, the component “SeatHeating” implements the operation “SetPower” and makes it available to other components through the port “Setting”. The component “SeatHeatingControl” uses the operation “SetPower” and expects such an operation to be available through the port “HeatingElement”.

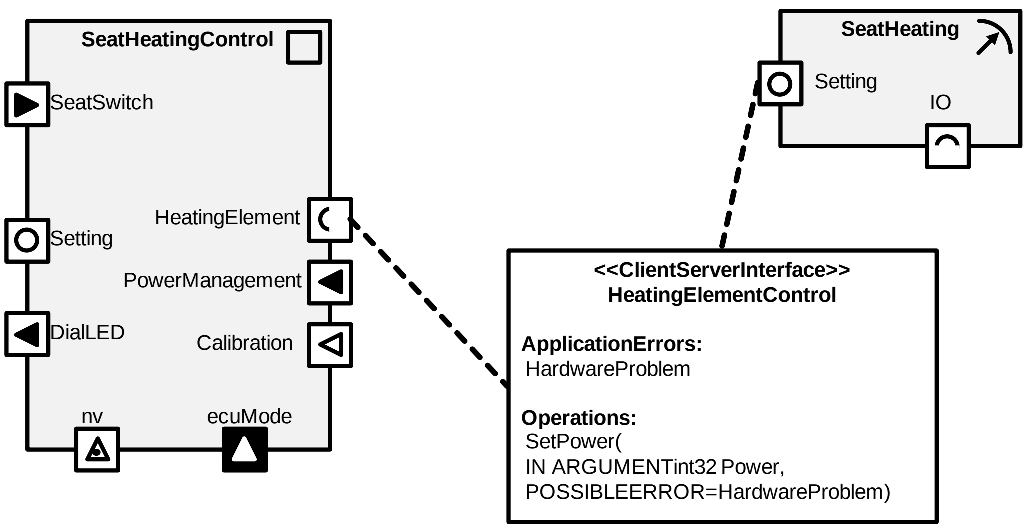


**Figure 3.6: Example showing the use of the Client-Server Interface “HeatingElementControl” to type the Port ”HeatingElement” of the component “SeatHeatingControl” and the port “Setting” of the component “SeatHeating”**

A component providing a sender-receiver interface generates values for the data-elements defined in the interface.
In the example of Figure 3.7, the component “SeatSwitch” 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”.

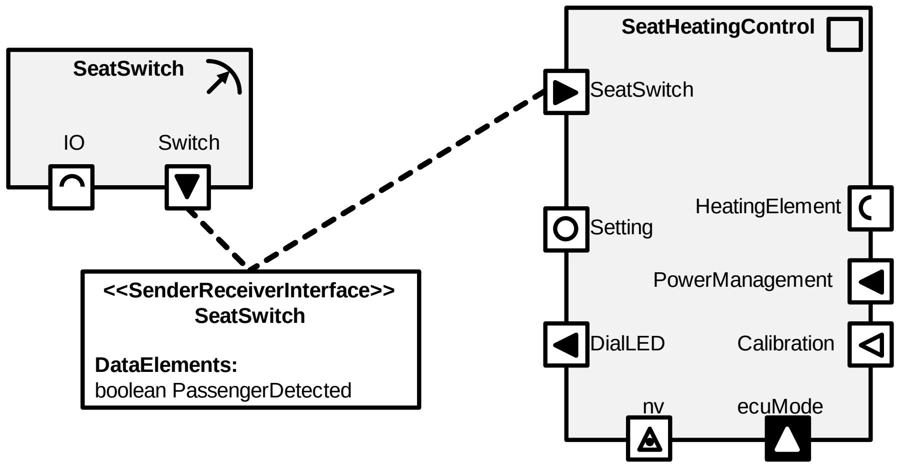


**Figure 3.7: Example showing the use of the Sender-Receiver Interface "SeatSwitch" to type the Port "SeatSwitch" of the components "SeatHeatingControl" and the port "Switch" of the component "SeatSwitch"**

## 3.4 Connectors

During the design of an AUTOSAR system, ports of components that need to communicate with each other are hooked up using assembly-connectors. Such an assembly-connector connects one RPort with one PPort.

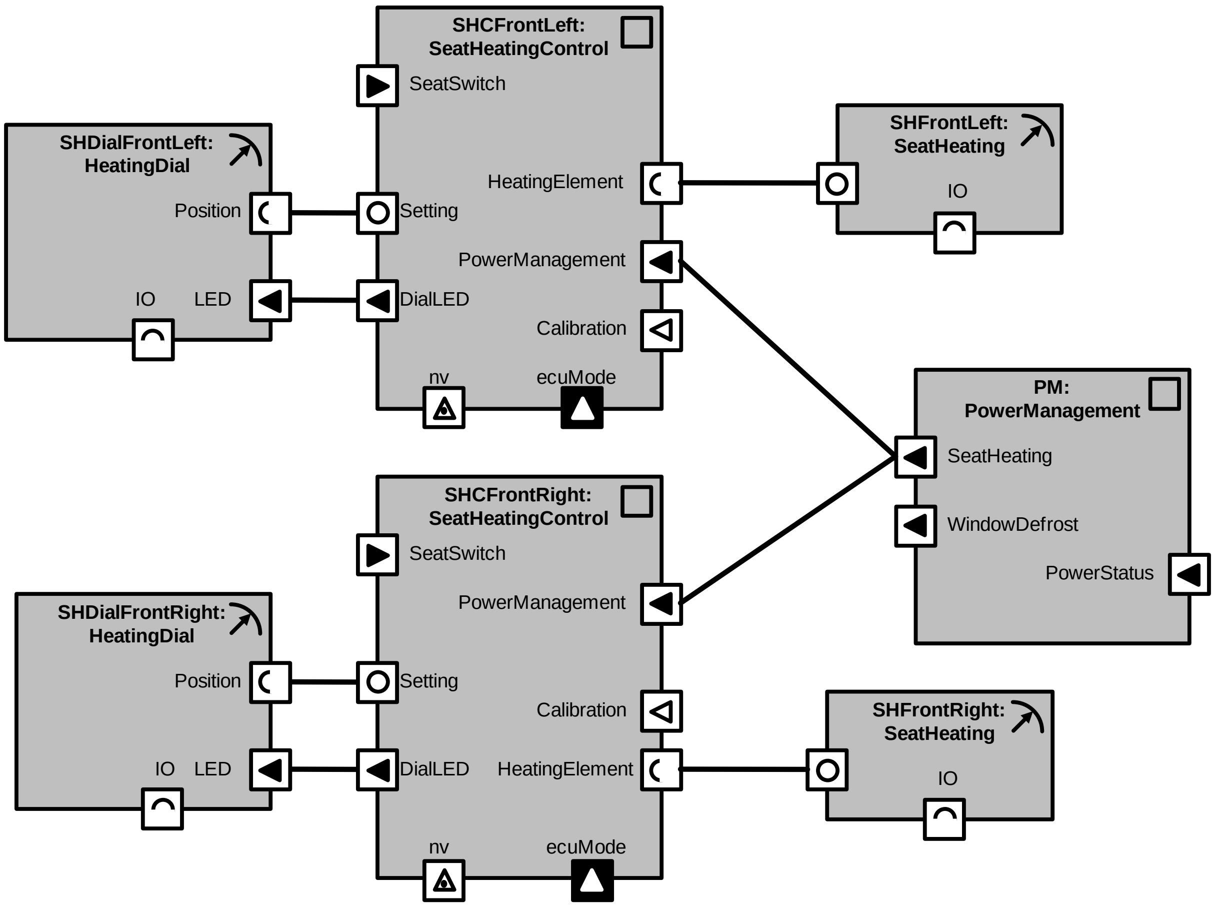


**Figure 3.8: Example of the use of eight assembly-connectors to connect the ports of seven components**

For the case of sender-receiver communication, the presence of an assembly-connector represents the fact that the data generated by the PPort on the connector is transmitted to the RPort. In the example of Figure 3.8 the data generated on the PPort "DialLED" of the component "SHCFrontRight" (of component-type "SeatHeatingControl") is transmitted to the RPort "LED" of the component "SHDialFrontRight" (of component-type "HeatingDial").
For the case of client-server communication, an invocation of the operations provided on a PPort is possible from the components that have an RPort connected to this PPort. In the example of Figure 3.8: when the component "SHDialFrontLeft" invokes an operation through the port "Position", this operation will be invoked on the port "Setting" of the component "SHCFrontLeft".
Both for sender-receiver communication and for client-server communication, one PPort can be connected to one or more RPorts (for multicast sending and multiple clients connected to a server, respectively). In the example of Figure 3.8, the data coming out of the port "SeatHeating" of the component "PM" is sent to both components "SHCFrontLeft" and "SHCFrontRight".
Furthermore, in sender-receiver communication one or more PPorts can be connected to one RPort (e.g. for information collected from different senders in a single receiver).
The exact communication behavior that such a connector represents depends on the kind of operations or data that is provided and/or required on the ports that the connector connects.

VFB008: At configuration time, all components instantiated on the VFB are known

VFB009: At configuration time, all communication possibilities between components on the VFB are modeled through the presence of connectors. Communication between ports not connected through such a connector is not possible.[5]

VFB010: An assembly-connector connects exactly one PPort with exactly one RPort

VFB113: An assembly-connector can connect one PPort with one RPort only if their port types, interfaces and attributes, characterizing their communication abilities, are compatible with each other[6].

## 3.5 Compositions versus atomic components

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”. A composition is itself a component-type and can have its own ports. Compositions can be used as structuring elements to build up hierarchical systems with an arbitrary number of hierarchies.
Figure 3.9 shows the definition of the composition “SeatHeatingControlAndDrivers”. This composition contains three prototypes: the prototype “SHDial” (of component-type “HeatingDial”), the prototype “SHC” (of component-type “SeatHeatingControl”) and the prototype “SH” (of component-type “SeatHeating”). The composition itself is a component-type and has seven ports.

[5] The AUTOSAR-Services are an exception to this rule. The connections related to AUTOSAR-Services are made later in the AUTOSAR-method, namely during ECU-configuration. See AUTOSAR Services, for a deeper explanation.
[6] The exact meaning of “compatibility” is defined in the [Software Component Template].

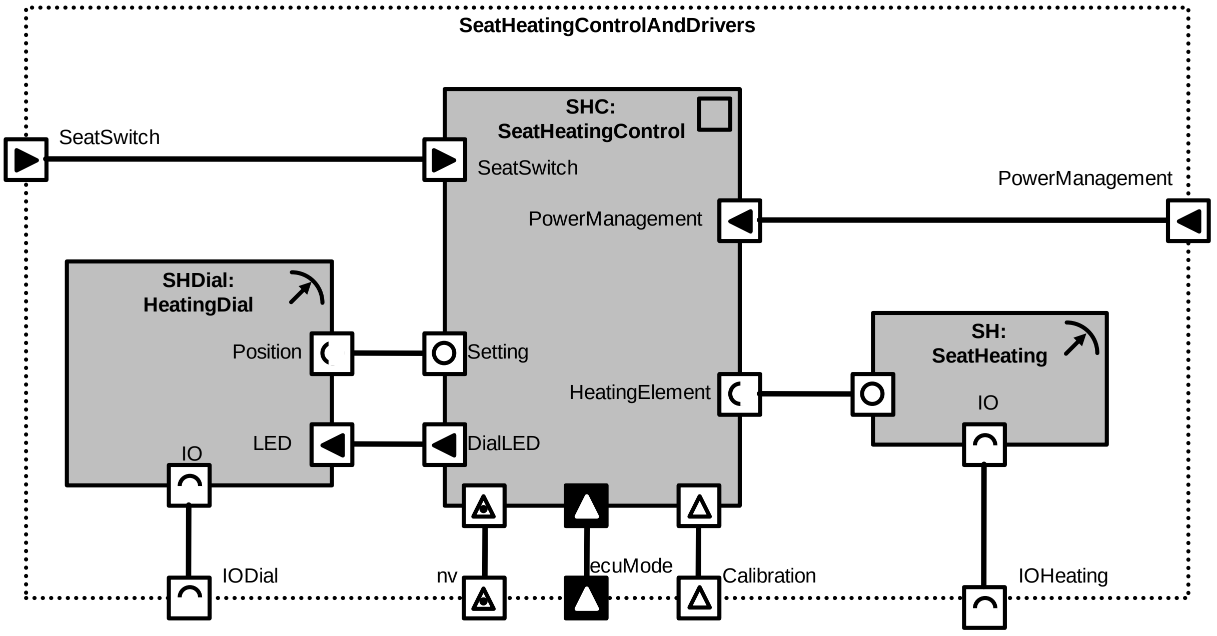


**Figure 3.9: Example of the definition of the Composition “SeatHeatingControlAndDrivers”**

Figure 3.10 shows the use of a composition as a component-type. Figure 3.10 essentially shows another composition containing three prototypes: the prototypes “SHFrontLeft” and “SHFrontRight” (both of type “SeatHeatingControlAndDrivers”) and the prototype “PM” of type “PowerManagement”.
A component-type in AUTOSAR is either a “composition” or “atomic”. A composition is defined through interconnected prototypes (as in Figure 3.9). An atomic component cannot be further decomposed into smaller components.

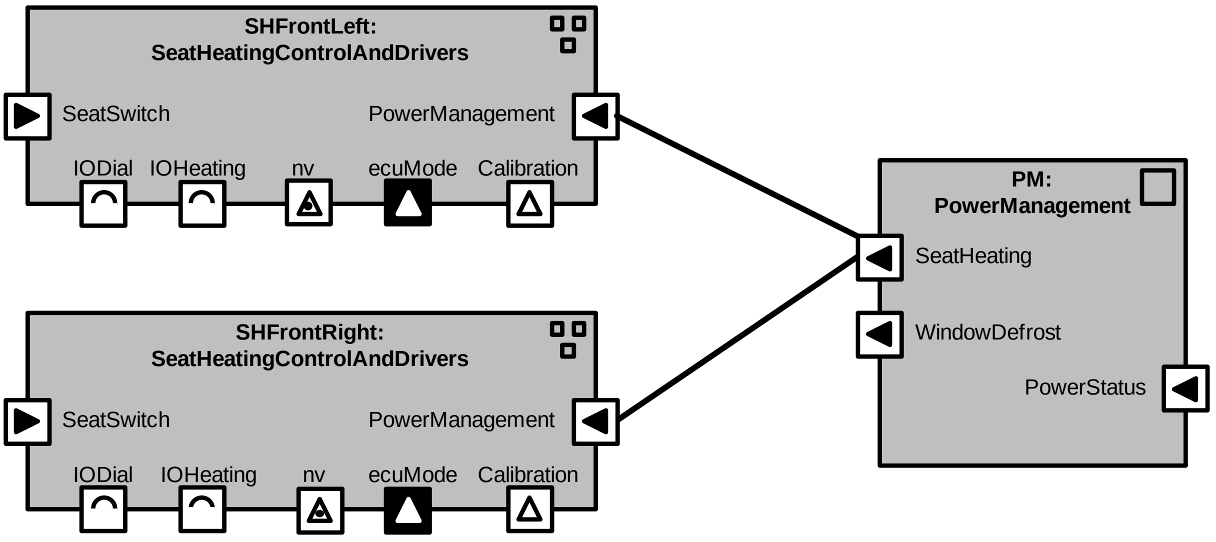


**Figure 3.10: Example of the use of the Composition “SeatHeatingControlAndDrivers”**

## 3.6 Relationship between the VFB and the ECU Software Architecture

When a sub-system consisting of atomic components and assembly-connectors is deployed on a network of ECUs, all atomic components are mapped on an ECU. The corresponding connectors between the components are implemented by intra- or inter-ECU communication mechanisms.
In the example of Figure 3.11, atomic components “SHDialFrontLeft” and “SHCFrontLeft” are mapped onto “ECU1”, whereas the atomic component “PM” is mapped onto “ECU3”. This implies that the connectors between the first two components are handled within ECU1, whereas the connection between the component “SHCFrontLeft” and the component “PM” will run through a network connection between ECU1 and ECU3.

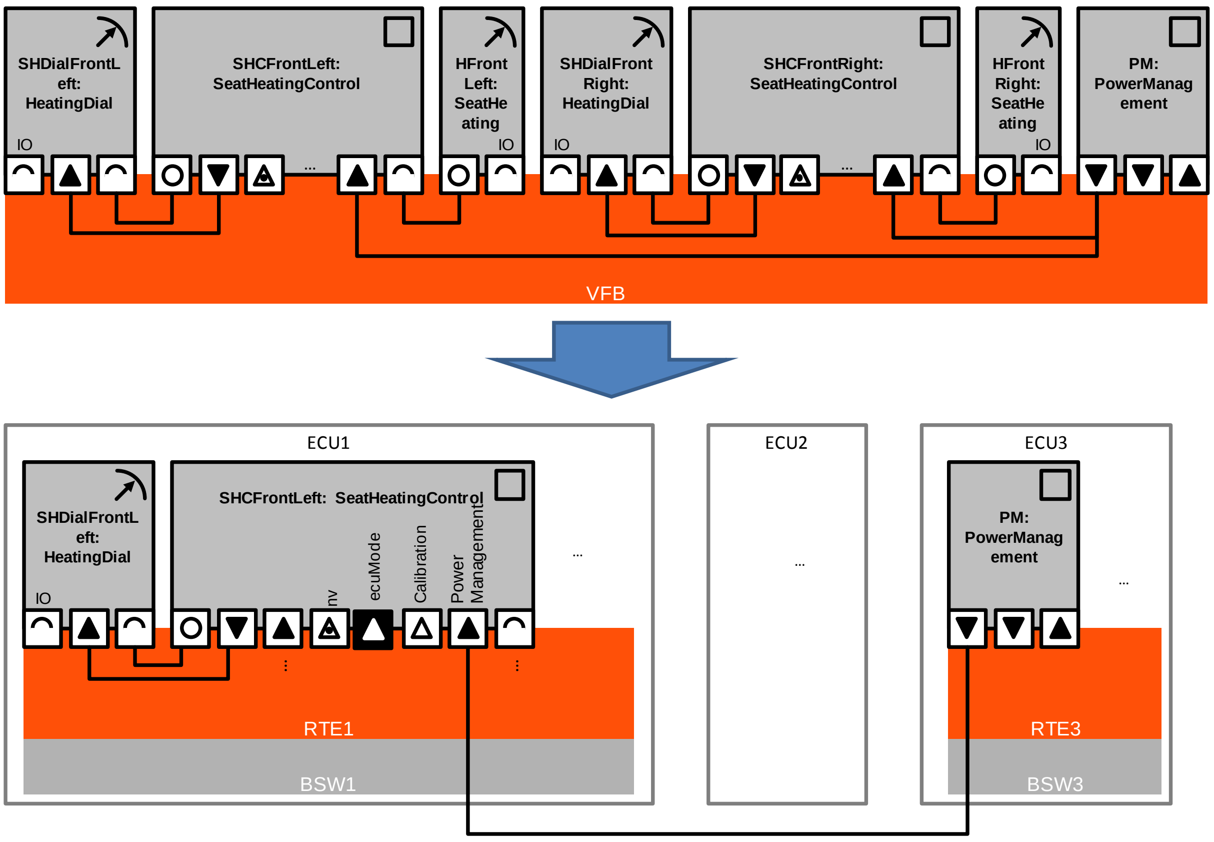

SHDialFrontLeft: HeatingDial
IO
SHCFrontLeft: SeatHeatingControl
HFrontLeft: SeatHeating
IO
SHDialFrontRight: HeatingDial
IO
SHCFrontRight: SeatHeatingControl
HFrontRight: SeatHeating
IO
PM: PowerManagement
VFB
ECU1
SHDialFrontLeft: HeatingDial
IO
SHCFrontLeft: SeatHeatingControl
nv
ecuMode
Calibration
Power Management
RTE1
BSW1
ECU2
ECU3
PM: PowerManagement
RTE3
BSW3

**Figure 3.11: Example illustrating the mapping of a composition of components on three ECUs.**

Figure 3.12 shows the standard component-view on the AUTOSAR layered software architecture, which is the architecture of a single AUTOSAR ECU. The “AUTOSAR Interface” of a component refers to the full set of ports of a component (as defined before, a port-interface characterizes a single port of a component). A “Standardized AUTOSAR Interface” is an AUTOSAR Interface which is standardized by AUTOSAR. Typically, an AUTOSAR service will have such a “Standardized AUTOSAR Interface”. For a formal definition of the term AUTOSAR Interface and Standardized AUTOSAR Interface see [Layered Software Architecture].

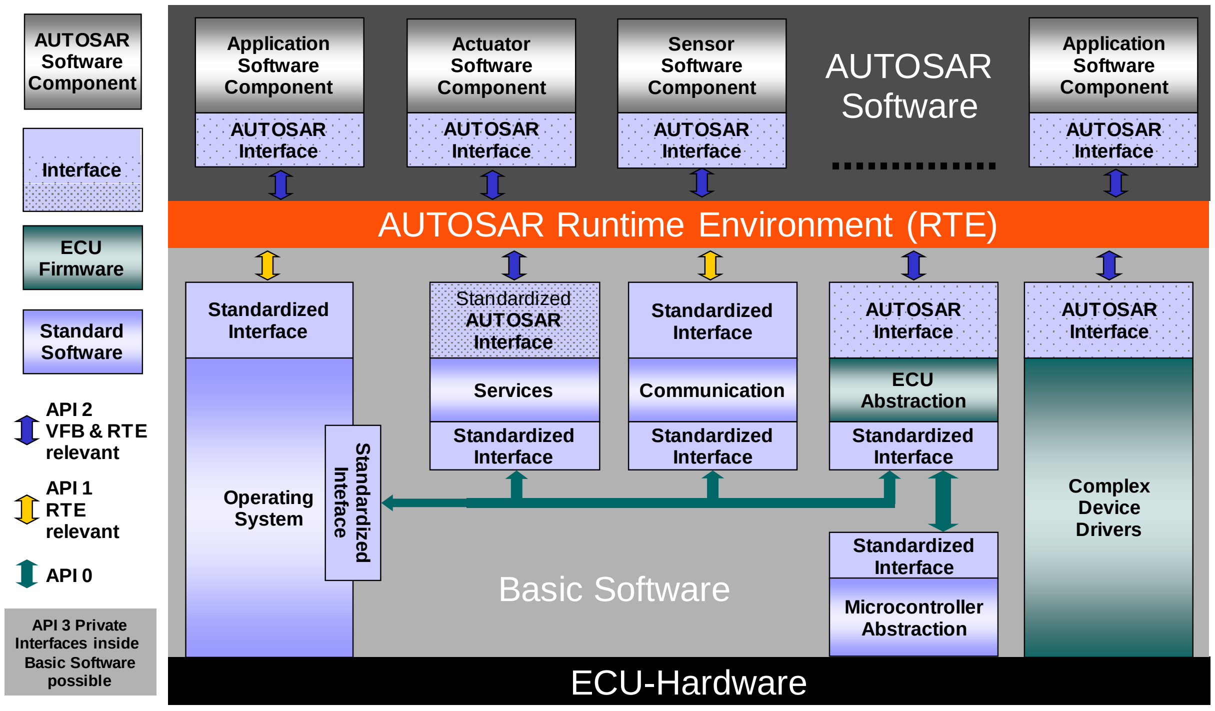


**Figure 3.12: Component-View on the AUTOSAR layered software architecture**

Figure 3.13 shows what a possible concrete architecture of ECU1 out of the example of Figure 3.11 might look like. The atomic software components that are mapped on ECU1 are hooked into the Run-Time Environment that is generated for ECU1. This Run-Time Environment will typically implement the local connections between the local components “SHCFrontLeft” and “SHDialFrontLeft”.
In addition, the Run-Time Environment has the responsibility to route information that is coming from or going to remote components. In the example, the port “Power Management” is routed to the communication stack in the underlying basic software.
The RTE also hooks up the component “SHCFrontLeft” to local standardized AUTOSAR services, such as the local non-volatile memory (through the port “nv”) and information on the local state of the ECU (“through the port “ecuMode”).

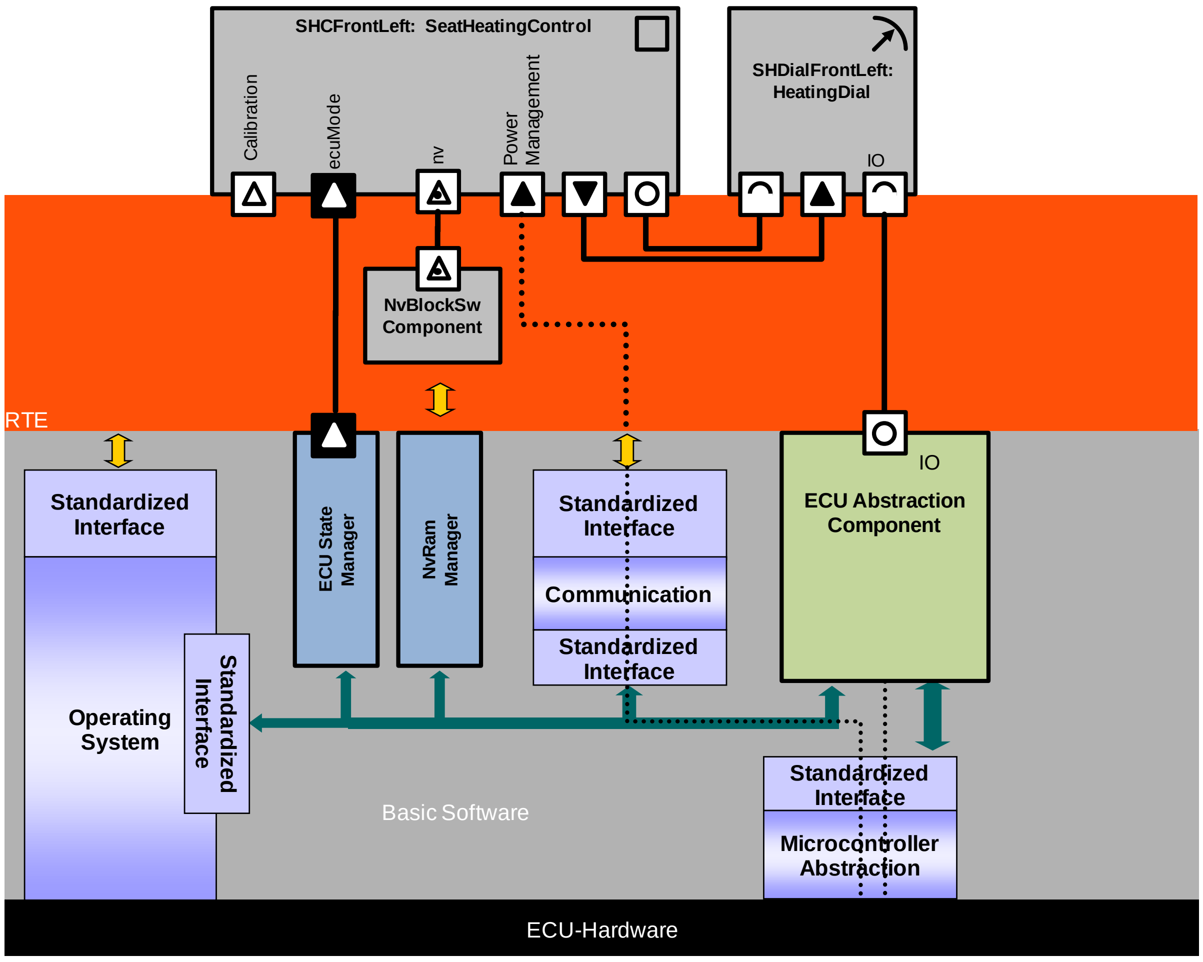
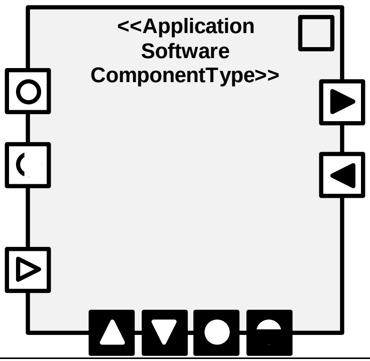
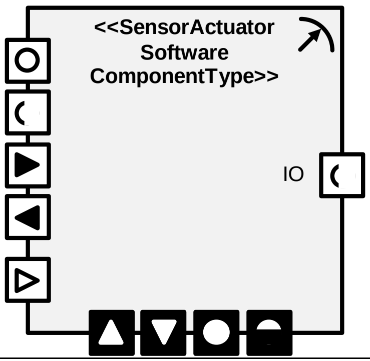
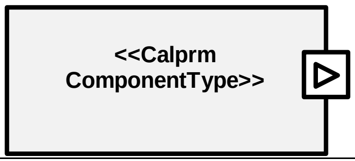
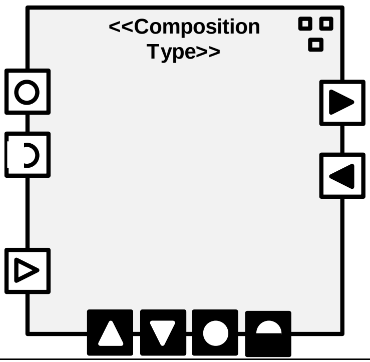
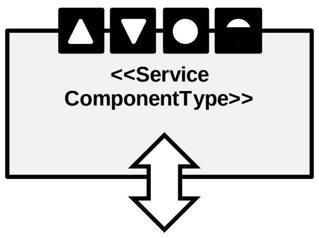
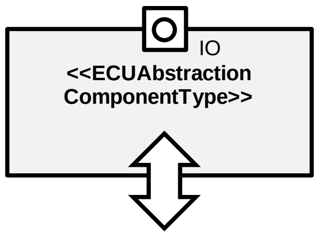


**Figure 3.13: Example showing the relationship between the components mapped on an ECU and the ECU Software Architecture**

## 3.7 Kinds of components

This section gives a final overview of the various kinds of components that are relevant to AUTOSAR.

| Kind | Description | Illustration |
|---|---|---|
| Application software component | The application software component is an atomic software component that implements (part of) an application. The atomic software components can use all AUTOSAR communication mechanisms and services. The application software component interacts with sensors or actuators through a sensor-actuator software component. |  |

| | | |
|---|---|---|
| Sensor-actuator software component | The sensor-actuator software component is an atomic software component that handles the specifics of a sensor and/or actuator. It directly interacts with the ECU-Abstraction (this is illustrated by a port called "IO"). See Chapter 6, Interaction with hardware. |  |
| Calibration parameter component | A calibration parameter component provides values for calibration parameters. See Chapter 9. . |  |
| Composition | A composition is defined through the interconnected component-prototypes it contains. Consequently it can use all AUTOSAR communication mechanisms and services. |  |
| Service component | A service-component provides standardized services through standardized interfaces. To provide these services, this component may interact directly with certain other basic-software modules (this is represented by the double arrow). See Chapter 7. |  |
| ECU-abstraction component | The ECU-abstraction provides access to the ECU's specific IO capabilities. These services are typically provided through client-server PPorts and are used by the sensor-actuator software components. The ECU-abstraction may directly interact with certain other basic-software modules (this is represented by the double arrow). See Chapter 6, Interaction with hardware. |  |

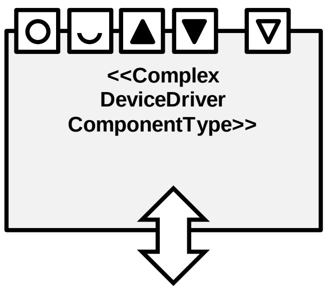
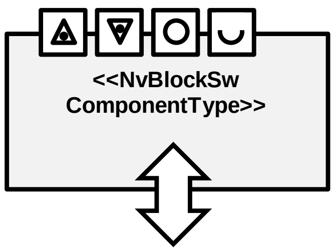
| | | |
|---|---|---|
| Complex device driver component | The complex device driver generalizes the “ECU-abstraction component”. It can define ports to interact with other components in specific ways and can also interact directly with other basic-software modules. The purpose of the complex device-driver is described further in Section 6.5 Complex Device Driver. |  |
| NVBlock software component | The NV Block Software Component allows SWC-S access to non volatile data. Specifically this block allows for the modeling of the NV data at the VFB level. It is the responsibility of the NV Block to map individual NV data elements to NV Blocks and to interact with the NV Manager in the BSW. The behavior of this component is to be generated based on the port services in the RTE. |  |

**Table 3.3: Kinds of components**

## 3.8 Resources for components and “runnables”

### 3.8.1 Background

The VFB is a system modeling and communication concept, which allows components to be distributed in a network of ECUs. The interaction possibilities between a component and other components are described through the component's ports and their associated interfaces, which define the operations, data-elements, mode-groups or calibration parameters that are provided or required by the component. Through the same communication mechanisms, the component can interact with standardized AUTOSAR services (available on each properly configured AUTOSAR ECU) or the ECU-specific IO capabilities (available on the specific ECU on which the appropriate hardware is present and to which the correct devices are connected).
However, implementations of components need access to additional resources, mainly memory (the component’s implementation typically needs memory to maintain its internal state) and CPU-power (the component’s implementation contains code that must be executed according to a certain timing schedule or in response to certain events).

As these scheduling issues are closely linked to the communication needs of the component, the RTE must provide both aspects. Therefore, the RTE must provide a complete environment for the component, including:

- Appropriate mechanisms through which the component’s implementation (for example in a programming language like “C”) can:
  - Provide values for data-elements in the component’s PPorts
  - Read/Consume values for data-elements in the component’s RPorts
  - Access the component’s calibration parameters
  - Provide implementations for the operations in the component’s PPorts
  - Invoke operations provided by other components through the component’s RPorts
  - Etc.
- Appropriate mechanisms through which the component’s implementation (for example “C” functions) is invoked in response to:
  - Fixed-time schedules (for example: many components need to run “cyclically”)
  - Events related to the communication mechanisms (for example some components might want to be notified upon the reception of data from other components)
- Appropriate mechanisms through which the component’s implementation can access other common resources, such as instance-specific memory
- As an AUTOSAR ECU typically is a multi-threaded environment, the RTE must also provide all common synchronization mechanisms

This section introduces the AUTOSAR construct that addresses these various needs: the “runnable”.

### 3.8.2 The “runnable” concept

The “atomicity” of an atomic software-component refers to the fact that the component cannot be divided in smaller components and must therefore be mapped onto a single ECU.
For example, Figure 3.14 shows a logical component view of the mapped application-software component “SHCFrontLeft” on a specific ECU. Through its ports, the component has expressed which information it requires and provides from other components.

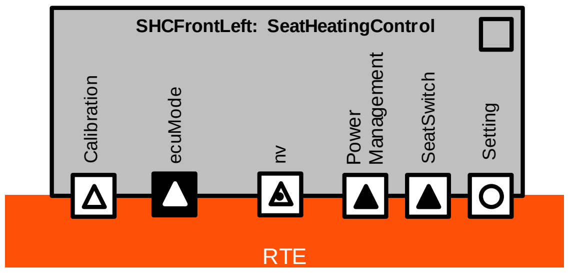


**Figure 3.14: Component-view on the interaction between an atomic software component and the RTE on an ECU**

However, the actual implementation of a component consists of a set of “runnable entities”[7] (also more simply called “runnables”). A “runnable entity” is a sequence of instructions (provided by the component) that can be started by the Run-Time Environment[8].

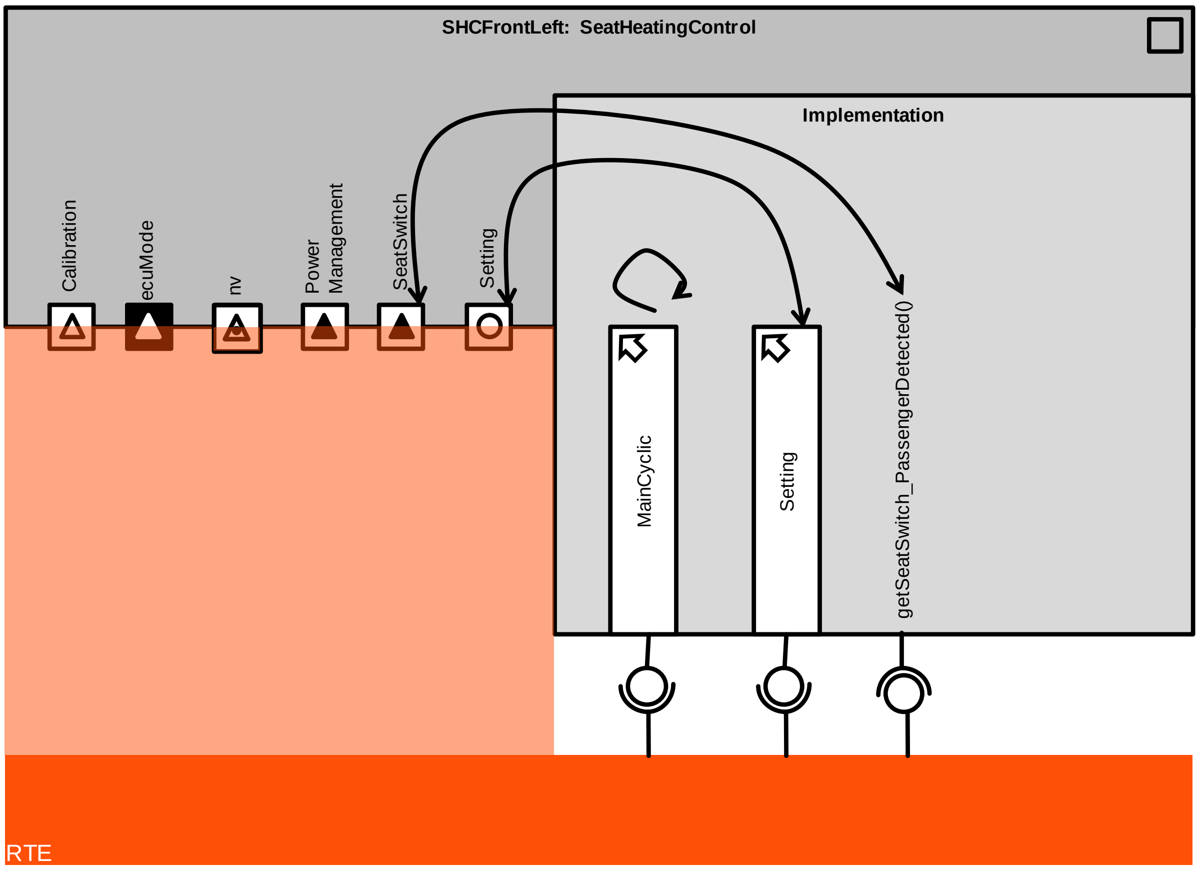


**Figure 3.15: Implementation-view on the interaction between an atomic software component and the RTE on an ECU**

Figure 3.15 shows an example of this. Logically, the component-type “SeatHeatingControl” has defined six ports, through which it wants to interact with other components or services. The implementation of the component on the other hand contains two runnables: “MainCyclic” and “Setting”. The component requires the runnable “MainCyclic” to be invoked cyclically (at a specific rate) by the RTE. The component requires that the second runnable “Setting” is invoked whenever another component invokes an operation on the PPort “Setting”. To access the information provided to the component through the RPort “SeatSwitch”, the component will use the operation “getSeatSwitch_PassengerDetected()” of the sender-receiver interface of the SeatSwitch port. The implementation of this interface is provided by the RTE.

[7] The usage of the word “runnable” is for example consistent with the “Runnable” Interface in Java: “the Runnable Interface should be implemented by any class whose instances are intended to be executed by a thread”.
[8] In certain cases, optimization of the RTE could cause a runnable entity to be started directly from another software-component without real intervention of the RTE. For example a synchronous call to a component that runs on the same ECU and can execute within the context (task) of the caller could be implemented as a direct function-call into the calling component.

In general, an atomic software-component can provide just one runnable or it can contain a large number of runnables. A runnable can be a very simple piece of code that executes a simple algorithm or a complex program.

VFB043: At configuration time, the runnables of a component must be known

A “runnable entity” runs in the context of a "task"[9]. The task provides the common resources to the “runnable entities” such as a context and stack-space. Typically the operating-system scheduler has the responsibility to decide during run-time when which “task” can run on the CPU (or multiple CPUs) of the ECU. There are many standard strategies that schedulers can use (e.g. priority-based preemptive, round-robin, time-triggered…).

### 3.8.3 The implementation of a component and the role of the RTE

In conclusion, the implementation of an atomic software-component essentially consists of three aspects:
A model of the component (using the concept of ports and port-interfaces) that is used to hook up the component with other components at the VFB-level
An implementation (“code”). The implementation of the component is structured in “runnables” which are pieces of code that can be executed by the RTE
A software-component description ([Software Component Template]) in which the component describes requirements on the RTE. These include:

- Which runnables need to be called cyclically
- Which runnables need to be called in response to events related to communication or other sources
- How the component would like to access the information in its ports or invoke the operations that it requires from other components
- Any other resources the component requires, such as AUTOSAR services or local memory

In a properly configured AUTOSAR ECU, the RTE (in cooperation with a properly configured basic software), will satisfy the component’s requirements. The RTE will for example:

- Ensure that the runnables are invoked at the correct times
- Provide the functions that the component needs to access data or invoked operations
- Provide all other resources the component needs

### 3.8.4 Categories of Runnables

Runnables are subdivided into the following categories:

1A) The Runnable uses implicit data access only and cannot block.

[9] Within this discussion, it is not necessary to make a distinction between “processes” (heavy-weight tasks which are often protected from other processes through memory-management) and “threads” (light-weight tasks running inside a process). The “task” refers to both.

1B) The Runnable may additionally use explicit reading and writing and cannot block. (Implicit read/write is also allowed)
2) The Runnable may use explicit reading/writing including blocking behavior.

Implicit data access means that the runnable does not actively initiate the reception or transmission of data. Instead, the required data is received automatically by the RTE when the runnable starts and provided data is made available for other runnables at the earliest when it terminates.
Explicit reading and writing means that a runnable employs an explicit API call to send or receive certain data.

# 4 Communication on the VFB

## 4.1 Introduction

This section specifies the communication mechanisms of the VFB, which atomic software components can use to communicate with each other.
Section 4.2, Error types, defines the types of errors that can appear in both Sender-Receiver and Client-Server communication models.
Section 4.3, Sender-Receiver communication, defines the functional semantics of sender-receiver communication in more detail. This section also defines the communication attributes that define the exact characteristics of the communication patterns provided by AUTOSAR. Some details related to mode-switches are covered in Chapter 8, Mode Management.
Section 4.4, Client-Server communication, does the same for client-server.

## 4.2 Error types

Errors are divided into two simple classes: infrastructure errors and application errors.
Infrastructure errors are returned when the infrastructure between the sender and the receiver, for sender-receiver communication, or between the client and the server, for client-server communication, failed. A typical example of an infrastructure error is a timeout. In case the client does not receive a response from the server within a certain amount of time (because the communication channel between client and server is not available or a message was lost) a "time-out" infrastructure error is returned to the client. The possible infrastructure errors are standardized by AUTOSAR.
Application errors are application-specific and must be defined as part of the sender-receiver interface, for sender-receiver communication, or client-server interface, for client-server communication.

## 4.3 Sender-Receiver communication

The sender-receiver pattern enables the distribution of information where a sender distributes information to one or several receivers or a receiver receives information from several senders. Figure 4.1 gives an example how sender-receiver communication is modeled in the AUTOSAR VFB View.

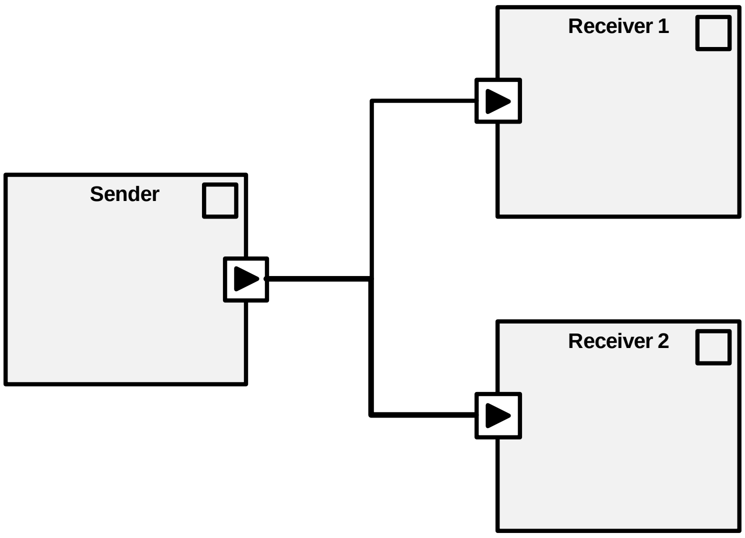


**Figure 4.1: Example of sender-receiver communication at VFB level**

In this example there are two assembly-connectors connecting the PPort of the component "Sender" with the RPort of "Receiver 1" (respectively "Receiver 2").
The sender-receiver interface associated with those ports consists of data-elements that define the data that is sent by the sender and received by the receivers.
The type of a data-element can be something very simple (like an "integer") or can be a complex (potentially large) data type (e.g. an array or a string). The transfer of a value, even of a complex data type, is always logically atomic.

| VFB011: At configuration time, the data-type of each data-element in a sender-receiver interface is known |
|---|

A sender can provide a new value for each data-element defined in the Sender-Receiver Interface. The precise semantics depend on whether the data-element is defined to be of type "last-is-best" or whether the data-element is "queued".

| VFB012: At configuration time, each data-element in a sender-receiver interface must be defined to have either "queued" or "last-is-best" semantics |
|---|

Each data-element with "last-is-best" semantics can be configured to support invalidation. If the "last-is-best" data-element supports invalidation, the sending component can indicate the receivers that the data-element is "invalid" (see attributes RECEIVE_INVALID and CAN_INVALIDATE in Table 4.1 and Table 4.2).

| VFB101: At configuration time, it must be known for each "last-is-best" data-element in a sender-receiver interface, whether the data-element supports the ability to be "invalid" or not |
|---|

### 4.3.1 From the point of view of the sender

Each data-element with "last-is-best"-semantics in a PPort of a sender-component always has a current value. The initial current value of such a data-element can be defined through configuration of the VFB (see attribute "INIT_VALUE" in Table 4.1

and in Table 4.2). The sending component can change the current value of the data-element, thereby overwriting the previous value of the data-element.
When a data-element has "queued" semantics, the consecutive values produced by the sender are stored in a queue. The initial queue has length zero (no values are available). Each time the sender produces a new value, this value is added to the queue, until an arbitrary and configurable number of entries has been reached.
A sending component does not know the identity and the number of receivers. Its behavior is independent of the presence or absence of receivers. Sender-receiver communication allows for a strong decoupling between sender and receiver. The sender just provides the information and the receivers decide autonomously when and how to use this information. It is the responsibility of the communication infrastructure to distribute the information. In certain cases, however, the sending application wants to be notified when the expected quality-of-service of the communication system between the sender and its receivers is known to be violated (see attribute "TRANSMISSION_ACKNOWLEDGEMENT" in Table 4.1).

VFB103: At configuration time, it must be known for each data-element in a PPort of a component, whether the component wants to be informed on successful transmission or timed-out transmission

Table 4.1 gives an overview of the communication attributes that a sender can use to control the behavior of the sender-receiver communication pattern. These attributes are defined at the level of a single data-element or mode-group.

| Attribute/Feature Name | Realization in software-component template | Description | Kind of data-element or modeGroup | | |
|---|---|---|---|---|---|
| | | | data | event | mode |
| INIT_VALUE | attribute "initValue" of "UnqueuedSenderCompSpec" | This attribute defines the initial value of the data-element, seen by all receivers of this data-element. This initial value can be overwritten by the attribute INIT_VALUE on the receiver side. | required | not available[10] | not available[11] |
| CAN_INVALIDATE | attribute "canInvalidate" of "UnqueuedSenderComSpec" | In case this feature is used, the sender can invalidate a data-element. | optional | not available | not available |
| MODE_QUEUE_LENGTH | "queueLength" of ModeSwitchComSpec | This attribute defines the size of the input queue of the of mode switch notifications to a mode machine. | not available | not available | required |

[10] The initial condition of a queued data-element is the empty queue
[11] The initial mode is defined as part of the ModeDeclarationGroup

| | | | | | |
|---|---|---|---|---|---|
| IMPLICIT_SEND | “DataWriteAccess” | Normally, a sender must make an explicit function-call to send a data-element or change the current mode. “Implicit sending” means that a runnable can modify a data-element while it is running. After the runnable terminates, the RTE will make the latest value available to receivers of the data-element. | optional | not available | not available |
| TRANSMISSION_ ACKNOWLEDGEMENT | “TransmissionAcknowledgementRequest” with attribute “timeout” or “ModeSwitchedAckRequest” with attribute “timeout” | The sending component is informed when the data has been sent correctly OR when the mode switch has been executed by the RTE. If the timeout occurs before this acknowledgement, the sender is informed of an infrastructure error. | optional | optional | optional |
| IS_QUEUED | “isQueued” in “DataElementPrototype” | When this parameter is TRUE, the data-element is queued (=used for “events”). When this parameter is false, the data-element has “last-is-best” semantics. | FALSE | TRUE | not available |

**Table 4.1: Communication Attributes for a Sender**

Details can be found in [Software Component Template] and [Specification of RTE Software].

## 4.3.2 From the point of view of the receiver

A receiver can access the value of each data-element defined in the Sender-Receiver Interface associated with the RPort of the receiving component.
For a data-element that has “last-is-best” semantics, the receiver has access to the latest value of that data-element. Alternatively, the receiver is informed that the data-element is “invalid” (in case the data-element supports this feature). The receiver may have access to the livelihood of the data-element, whether its value is valid or outdated. The livelihood is defined by configuring the VFB (see attributes “TIME_FOR_RESYNC” and “ALIVE_TIMEOUT” in Table 4.2).

VFB014: At configuration time, the initial value of each last-is-best data-element in an RPort of a component must be defined

VFB015: The current value of a data-element seen by a receiving component, when a sending-component has not provided a value, is the configured initial value of the RPort

VFB017: The initial value of the receiving component can be “invalid” if the data-element supports this

VFB094: At configuration time, it must be known for each last-is-best data-element in a RPort of a component whether the component wants to get informed of the livelihood of the data-element

VFB095: A receiver that gets informed of the livelihood of a data-element must configure the period of time between receptions. This threshold determines the livelihood of the data-element: actual or outdated

For a data-element that has “queued” semantics, the receiver has essentially one operation: to obtain the next data-element from the queue. In case the queue is empty, this fact is returned to the receiver. Otherwise, the next data-element value is read and taken from the queue (in other words, this is a “consuming read”). The capacity of the queue is defined by configuring the VFB (see attribute “RECEIVER_QUEUE_LENGTH” in Table 4.2).

VFB019: The queue associated with a data-element with “queued” semantics is initially (before a sender has added values to the queue) empty

VFB020: Logically, the queue is located on the receiver’s side

VFB021: At configuration time, the size of the receiver’s queue must be known

VFB022: The receiver’s queue has first-in first-out semantics

VFB023: When the receiver’s queue is full and a new value arrives, this value is dropped (“queue overflow”)

VFB024: The receiver can be notified of “queue overflow” if it indicates that it desires this notification at configuration time

Table 4.2 gives an overview of the communication attributes that a receiver can use to control the behavior of the sender-receiver communication pattern. These attributes are defined at the level of a single data-element or mode-group.

| Attribute Name | Attribute Value | Description | Kind of data-element or modeGroup | | |
|---|---|---|---|---|---|
| | | | data | event | mode |
| INIT_VALUE | " initValue" of "UnqueuedReceiverComSpec" | A receiver can optionally specify its own initial value, which overrides the initial value of the sender. | optional | not available[12] | not available[13] |
| RECEIVE_INVALID | "handleInvalid" in "UnqueuedReceiverComSpec" | The receiver can specify how it wants to respond when an invalid value for a data-element is received. | optional | not available | not available |
| TIME_FOR_RESYNC | " resyncTime" of "UnqueuedReceiverComSpec" | Time allowed for resynchronization of data values after current data is lost, e.g. after an ECU reset. | optional | not available | not available |
| ALIVE_TIMEOUT | "aliveTimeout" of "UnqueudReceiverComSpec" | The receiver specifies the maximum period of time it may take to receive a data-element If the data-element is not received within the defined period, the data-element is "outdated" | optional | not available | not available |
| IMPLICIT_RECEIVE | "DataReadAccess" | Normally, a runnable wishing to read a data-element needs to do this through an explicit call to the RTE. The "IMPLICIT_RECEIVE" means that the runnable has access to the value of the data-element that was available at the time of the start of the runnable. It does not need to invoke an explicit API to fetch the latest data. | optional | not available | not available |

[12] The initial condition of a queued data-element is the empty queue
[13] The initial mode is defined as part of the ModeDeclarationGroup

| | | | | | |
|---|---|---|---|---|---|
| RECEIVE_EVENT | “DataReceivedEvent” and “ModeSwitchEvent” | This implies that the receiving applications is notified by the RTE when a new value of a data-element or a mode-switch is received. This implies that the receiving component does not need to poll but can wait for new data-elements or mode-changes. | optional | optional | optional |
| IS_QUEUED | “isQueued” in “DataElementPrototype” | When this parameter is TRUE, the data-element is queued (=used for “events”). When this parameter is false, the data-element has “last-is-best” semantics. | FALSE | TRUE | not available |
| RECEIVER_QUEUE_LENGTH | queueLength of QueuedReceiverComSpec | Received values are added to the end of the queue and values are read (consuming) from the front of the queue (i.e. the queue is first-in-first-out). If the queue is full and another data-item arrives this data item is discarded and the receiver is informed by error-handling mechanisms. | not available | required | not available |

| | | | | | |
|---|---|---|---|---|---|
| FILTER | Attribute " DataFilter" of "ReceiverComSpec" | A data-element is only passed to the application if the value of the data-element passes the conditions of the filter. If a newly received value for a data-element does not pass the conditions of the filter, the value is discarded (not added to queue for a queued receiver OR the current value of the data-element is not updated for a last-is-best receiver). The VFB provides the same filters as defined in OSEK-COM V3.0.3, P.12. These filters can only be applied to data-elements that are of a primitive type. | optional | optional | not available |

**Table 4.2: Communication Attributes for a Receiver**

Details can be found in [Software Component Template] and [Specification of RTE Software].

## 4.3.3 Multiplicity of sender-receiver

The term multiplicity discussed in the following two sections applies to the connection multiplicity of a specific port to one or more other ports; it does not concern two distinct ports of a software component that are connected separately to two distinct ports of another software component.
Both types of sender receiver semantics (i.e. an interface with data-elements of "last-is-best" semantics or queued semantics), support either 1:n communication (1 sender and n receivers, with $n \geq 0$) or n:1 communication (n senders and 1 receiver). The sender(s) own(s) the current value of the data-element. With last-is-best semantics the receiver(s) of the data always want(s) to have only the most recent value of the data. It is the responsibility of the communication system to ensure the availability of the correct value of the data-element on the receiver side. This is illustrated in Figure 4.2.

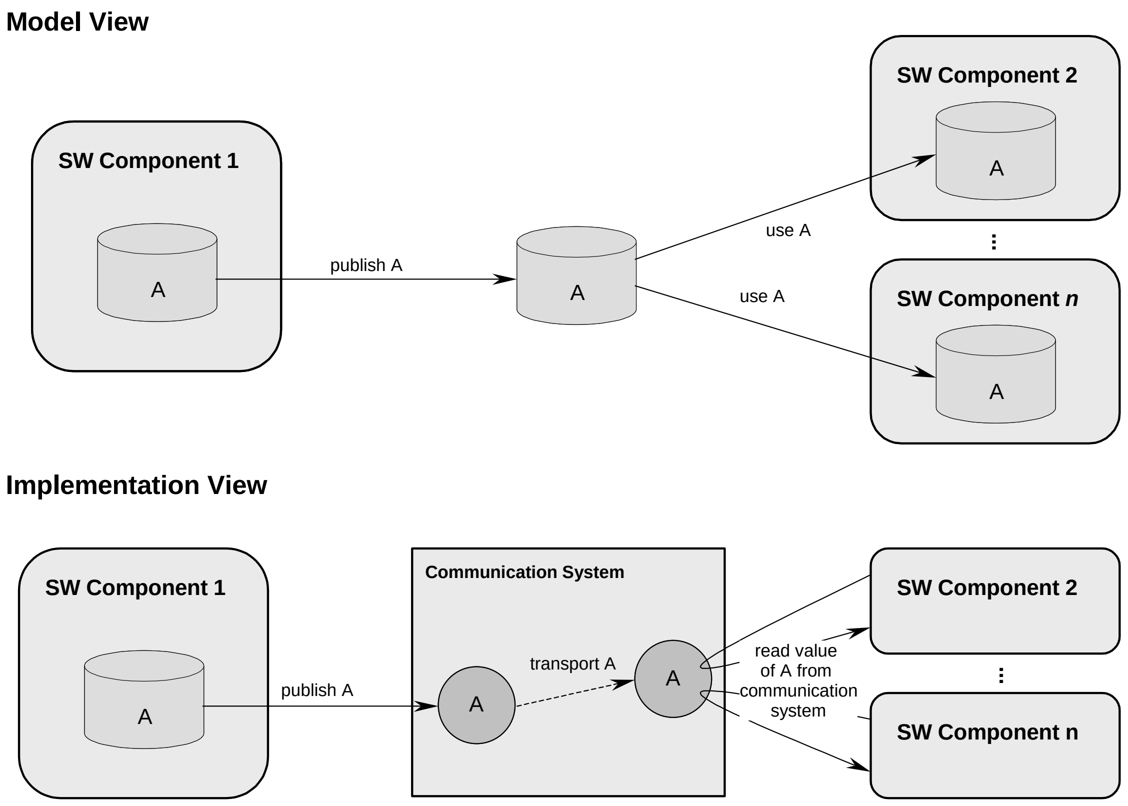


**Figure 4.2: "last-is-best" semantics. The upper part of this figure shows the model view of "last-is-best" semantics. The lower part shows the implementation view of this pattern.**

From an implementation point of view, this could for example be realized by having the sender periodically broadcast the latest value of the data-element to its receivers. A second implementation could only communicate actual changes to the receivers.
With "queued" semantics and n:1 communication the queue is on the receiving side and several senders can add values for the data-element to the single receiver's queue. To avoid a further increase of the complexity of the VFB mechanisms all other communication scenarios like n:m (n, m > 1) are not possible.

VFB025: For sender-receiver with data-elements with "last-is-best" semantics, both 1:n as well as n:1 communication (1 sender to multiple receivers) is possible

VFB026: For sender-receiver with data-elements with "queued" semantics, both 1:n (1 sender to multiple receivers) and n:1 communication (multiple senders to 1 receiver) is possible

VFB120: For sender-receiver with ModeDeclarationGroups, only 1:n (1 sender to multiple receivers) is possible

As a component can have an arbitrary number of ports, a single component can assume the role of sender and/or receiver.

### 4.3.4 Filtering between the sender and the receiver

The VFB supports the definition of an additional filter that sits between the sender and the receiver.
A new value for a data-element is only passed to the application if the value passes the conditions of the filter. If a newly received value for a data-element does not pass the conditions of the filter, the value is rejected (not added to queue for a queued data-element) or the current value of the data-element is not updated (for a last-is-best data-element).
The filters supported by AUTOSAR are the same as the filters, defined in OSEK-COM V3.0.3. These filters can only be applied to data-elements that are of a primitive type.

VFB027: At configuration time, the optional filter on the receiver's side must be defined

VFB028: The filter has the capabilities of the OSEK-COM V3.0.3 filter

In the VFB-model, such a filter can only be specified on the receiving side. This however, does not imply that the filtering should be implemented in the RTE on the receiving side. For example, consider the case that a receiving filter indicates that the receiver only wants to receive data-elements above a certain value, and that this is the only receiver hooked up to the sender over a network-connection. In that case a good implementation might decide to filter out the unnecessary values before they are sent onto the network (on the sending side).

### 4.3.5 Concurrency and ordering within a sender-receiver connector

Within the scope of a single connector between a sender's PPort and a receiver's RPort, the VFB preserves the order of the consecutive changes to the value of a specific data-element.

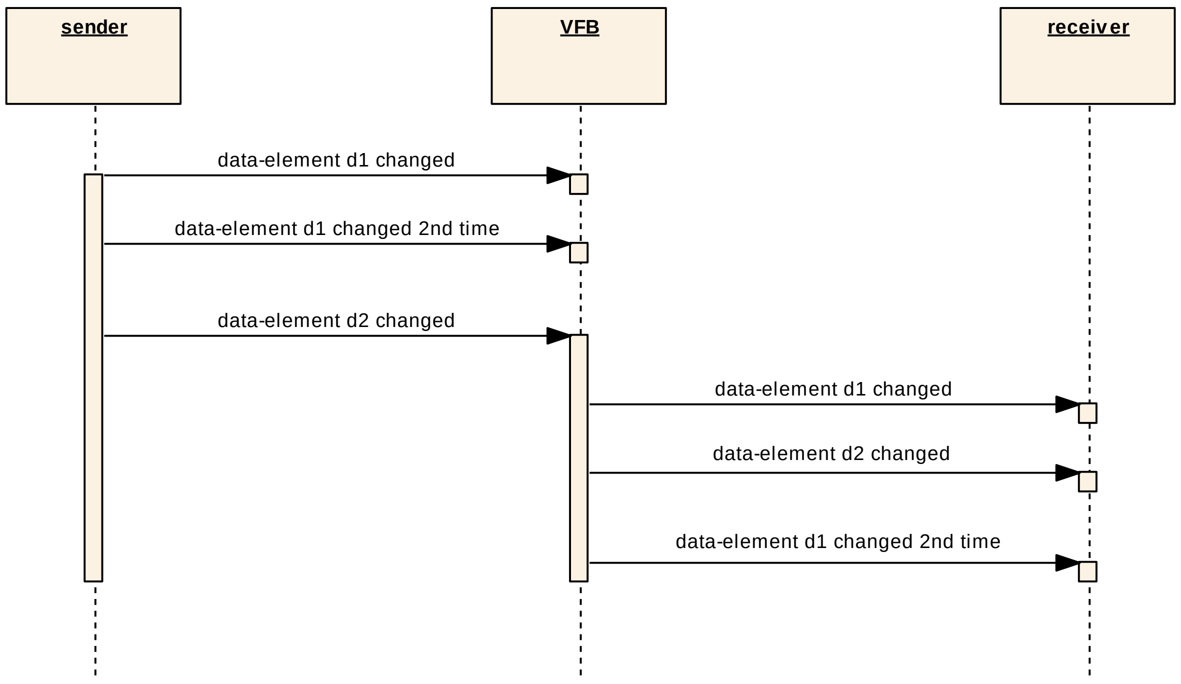


**Figure 4. 3: concurrency and ordering within a sender-receiver connector**

In the case of a queued data-element, the receiver must see the consecutive queued values of the data-element in the same order as the order in which they were produced by one specific sender.
In the case of "last-is-best" semantics, the semantics directly imply that "older" values should never overwrite "newer" values.
However, the VFB does not guarantee any ordering between changes to different data-elements (even not within the same interface) or between different connectors.
The VFB does not guarantee any ordering between mode switches of different ModeDeclarationGroups (even not within the same interface) or between different connectors.

VFB029: Within an individual sender-receiver connector, the VFB guarantees ordering in the changes made to an individual data-element

## 4.4 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[14] and the client is a user of a service. One simple example is the decoding of encrypted wireless key data (immobilizer, see Figure 4.4).

[14] Service in this chapter is a functionality which is offered by a certain AUTOSAR SW-component, the server, and which can be used by other AUTOSAR SW-component, the clients. It is not to be mixed up with an AUTOSAR service, defined more precisely in section 7, AUTOSAR Services.

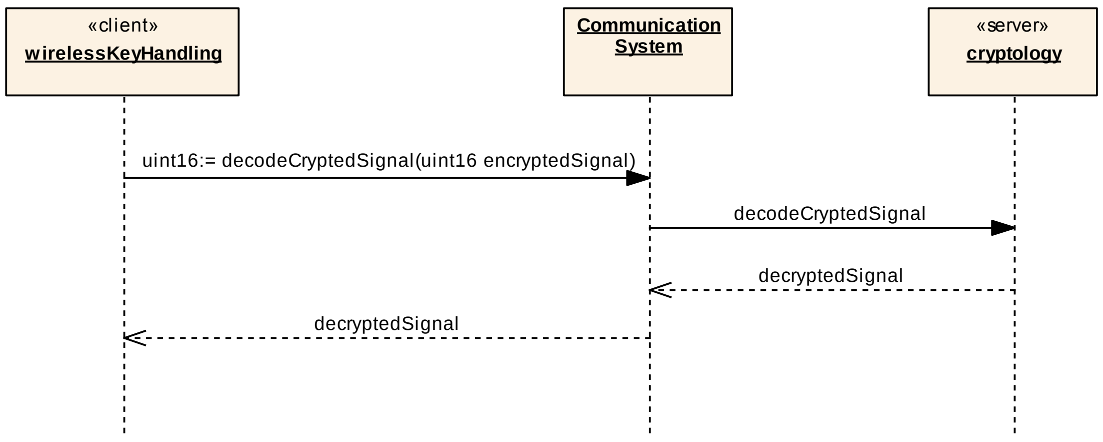


**Figure 4.4: Example of a synchronous client-server communication: decoding of encrypted wireless-key data (immobilizer).**

AUTOSAR defines a very simple, static n:1 client-server mechanism (n clients and 1 server, with n ≥ 0)[15]. Figure 4.5 gives an example how client-server communication for a composition of three components and two connections is visualized in the VFB View.

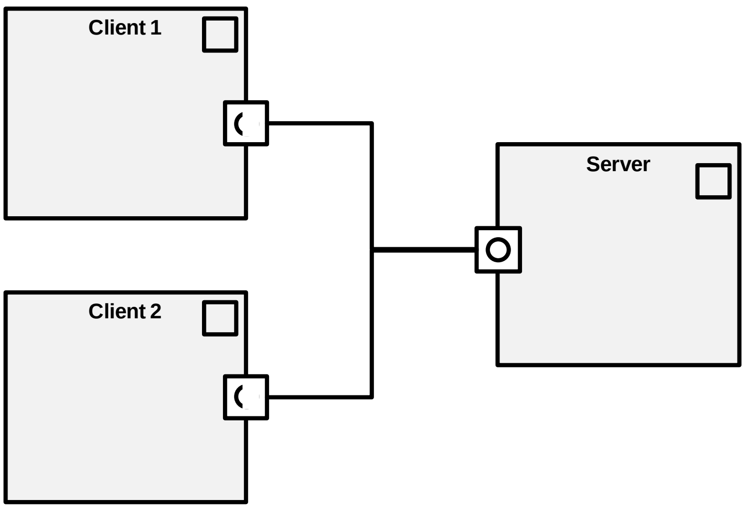


**Figure 4.5: Client-server communication in the VFB View**

In this example, there are 2 assembly-connectors. They hook up the RPort of "Client 1" (respectively "Client 2") with the PPort of the server. Each port is associated with a client-server interface, which defines the operations that are made available by the server and used by the client.
Each operation in such a client-server interface is associated with arguments, which are transported between the client and the server. These arguments are typed. The

[15] More complex client-server architectures might involve brokers that register services provided by servers and clients subscribing dynamically to certain services. To support the realization of such mechanisms, AUTOSAR could be extended by defining additional AUTOSAR Services (see section 7, AUTOSAR Services).

type of an argument in an operation could be a simple elementary data-type (like an integer in a certain range or a boolean) or complex structures or arrays.[16]

VFB031: At configuration time, for each operation in a client-server interface, the ingoing arguments, the returning arguments and their data-types must be known

Figure 4.6 illustrates the client-server mechanism through the VFB.

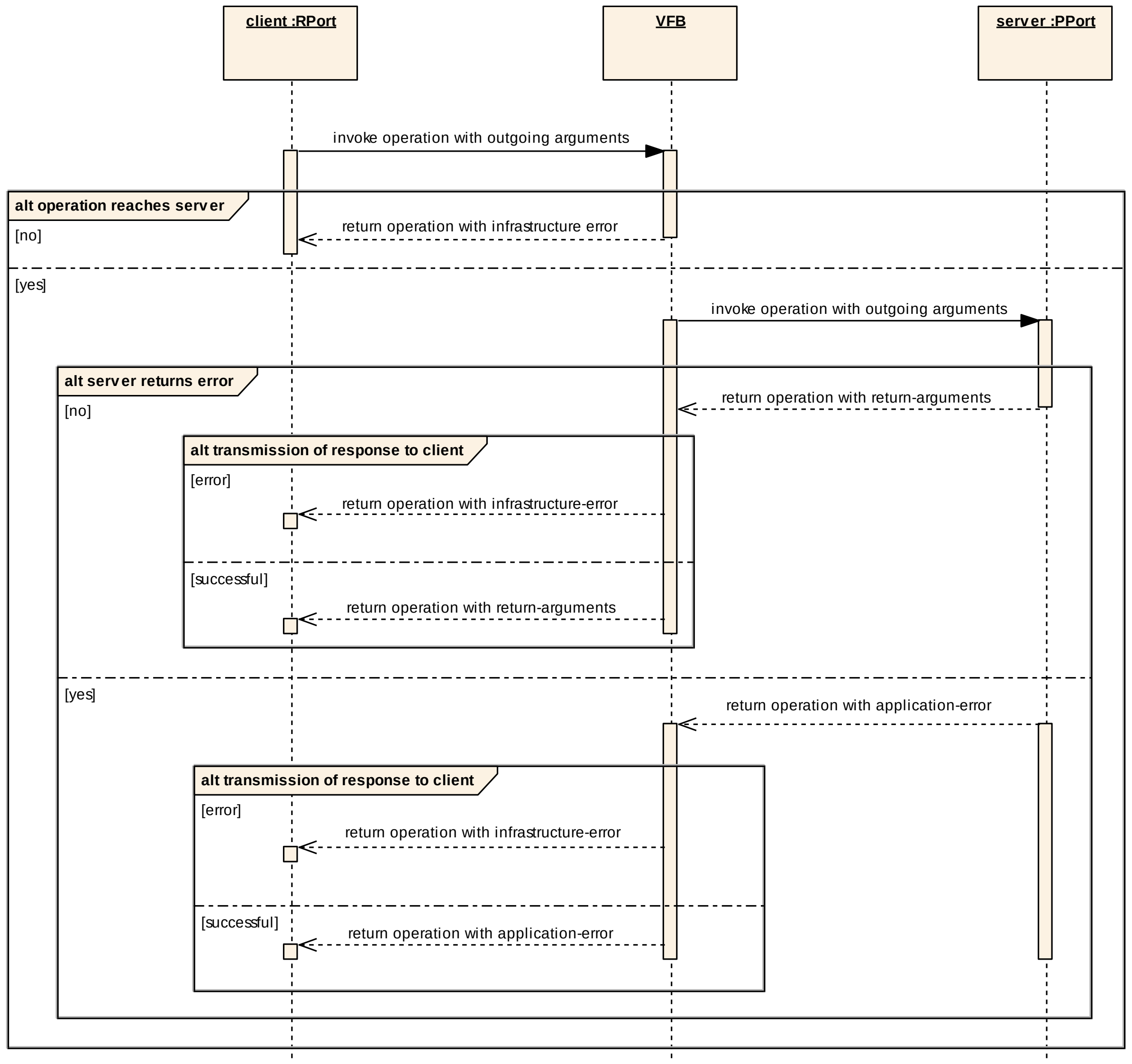

client :RPort
VFB
server :PPort
invoke operation with outgoing arguments
alt operation reaches server
[no]
return operation with infrastructure error
[yes]
invoke operation with outgoing arguments
alt server returns error
[no]
return operation with return-arguments
alt transmission of response to client
[error]
return operation with infrastructure-error
[successful]
return operation with return-arguments
[yes]
return operation with application-error
alt transmission of response to client
[error]
return operation with infrastructure-error
[successful]
return operation with application-error

**Figure 4.6: Client-server on the VFB (synchronous and asynchronous)**

[16] Details about the data-types supported by AUTOSAR in arguments can be found in [Software Component Template].

### 4.4.1 From the point of view of the client

The client initiates the client-server mechanism by requesting that the server performs a specific operation defined in the interface. The client thereby provides a value for each of the outgoing arguments defined for that operation in the Client-Server Interface.
Eventually, the client will either receive a valid response for the invocation or it will receive an error in response to the invocation of the operation. A valid response means that the server has executed the operation. In this case, the client receives a value for each return argument defined for the operation in the interface.
In case the operations change the state of the server, they should be designed carefully, so that the client can put the server easily in a known state or can simply repeat the operation in case of an infrastructure error. A good rule is to make the operation "idempotent", which means that an operation (with specific arguments) can be repeated an arbitrary number of times.

VFB032: A client can invoke an operation defined in a client-server interface of one of its RPorts

VFB033: When invoking an operation, the client must provide a value for each outgoing argument defined for that operation

VFB034: A client will receive exactly one response for each operation invocation

VFB035: The response which the client receives can be an infrastructure-error, an application-error or a valid server-response

VFB036: When the client receives a valid server-response, it obtains a value for each return-argument of the operation

VFB037: At configuration time, the possible application-errors that can be returned by the server to the client for the operation must be known

VFB038: The possible infrastructure-errors provided to the client as a possible response to a client invocation are standardized by AUTOSAR

Table 4.3 shows the communication attributes of a client.

| Attribute Name | Realization in software-component template | Description |
|---|---|---|
| CLIENT_MODE | Covered indirectly by the “SynchronousServerCallpoint”, the “AsynchronousServerCallpoint” and the “AsynchronousServerCallReturnsEvent” | The developer of a client can choose how to interact with the server.<br>In case the CLIENT_MODE is “synchronous”, the runnable invoking the operation is blocked until either a response has been received from the server, an infrastructure error is returned or the configured maximal blocking time expires. |

| | | |
|---|---|---|
| | | In case the CLIENT_MODE is "asynchronous - wakeup_of_wait_point" the runnable invoking the operation is not blocked. A runnable can wait for the response (from the server or because of an infrastructure error) in a wait-point.<br>In case the CLIENT-MODE is "asynchronous - activation_of_runnable entity", the runnable invoking the operation is not blocked. When the response (from the server or an infrastructure error) is available, a runnable is started which can process the response of the server |
| TIMEOUT | Attribute "timeout" of ServerCallPoint | Time in seconds before the server call times out and returns with an error message. How this infrastructure-error is reported depends on the call type (synchronous or asynchronous). |

**Table 4.3: Communication Attributes for a Client**

## 4.4.2 From the point of view of the server

A server waits for incoming invocations of operations from its clients. It performs the requested operation using the argument-values provided by the client. On finishing the execution of the requested operation, the server provides a value for each of the return-arguments to the client. In case the server encountered an error, it can alternatively return an application-error to the client instead of a set of values for the return-arguments.
Table 4.4 shows the communication attributes of a server.

| Attribute Name | Realization in software-component template | Description |
|---|---|---|
| QUEUELENGTH | Attribute "queuelength" of ServerCompSpec | On server side, there is a queue with length *n*, consuming reading and first-in-first-out strategy. If the queue is full, and another request arrives, the new request is discarded and the client will receive a "time-out" infrastructure error. |

**Table 4.4: Communication Attributes for Server**

### 4.4.3 Multiplicity of client-server

For client-server communication only “n:1”-communication (n clients, n>=0, 1 server) is supported.

VFB039: For client-server communication, only n:1-communication (n clients, 1 server) is supported

Each client RPort must be hooked up to exactly one connector, which links that RPort to exactly one PPort of a server. A PPort of a server on the other hand can be hooked up to an arbitrary number of client RPorts, i.e. none or more clients can invoke operations from the same server.  The implementation of the client-server communication has to ensure, that the result of the invocation of an operation is dispatched to the correct client.
As a component can have an arbitrary number of ports, a single component can assume the role of both client and server.

### 4.4.4 Ordering and concurrency within a client-server connector

A client is not allowed to invoke a specific operation on an RPort before the previous invocation of the same operation in the same RPort has returned (with either a valid response from the server or with an error).  This is illustrated in Figure 4.7.

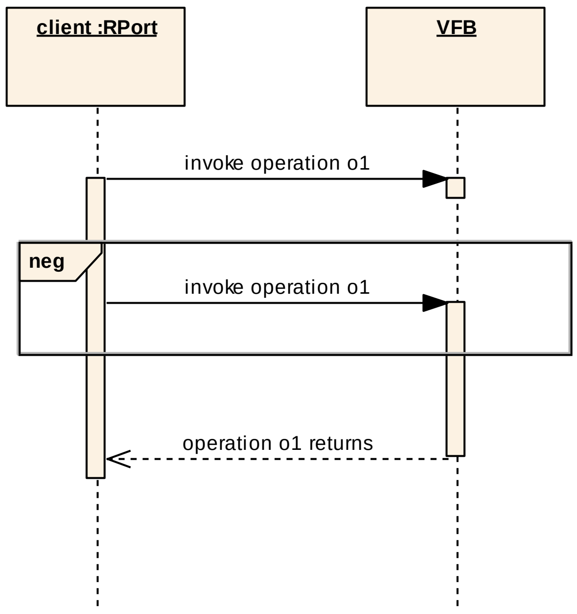


**Figure 4.7: Concurrent invocation of the same operation is not allowed**

The client is however allowed to make an invocation of a different operation on the same RPort before the invocation of a first operation has returned.  However, in this case, the VFB does not make any guarantees on the ordering of those invocations.
More specifically, it does not guarantee that the server sees the invocation of operations in the same order, as the order in which the client made those invocations.  Similarly, there is no guarantee that the responses are made available

to the client in any specific order (for example, in the order in which the client invoked those operations).
Although ordering is not guaranteed, the implementation of the VFB must make it possible for a client to associate a response from a server (or from the infrastructure in case an infrastructure-error is returned) with the correct corresponding invocation made by the client.

VFB040: A client is not allowed to invoke a specific operation on an RPort before the previous invocation of the same operation has returned

VFB042: It must be possible for a client to associate a response with the correct corresponding invocation made by the client

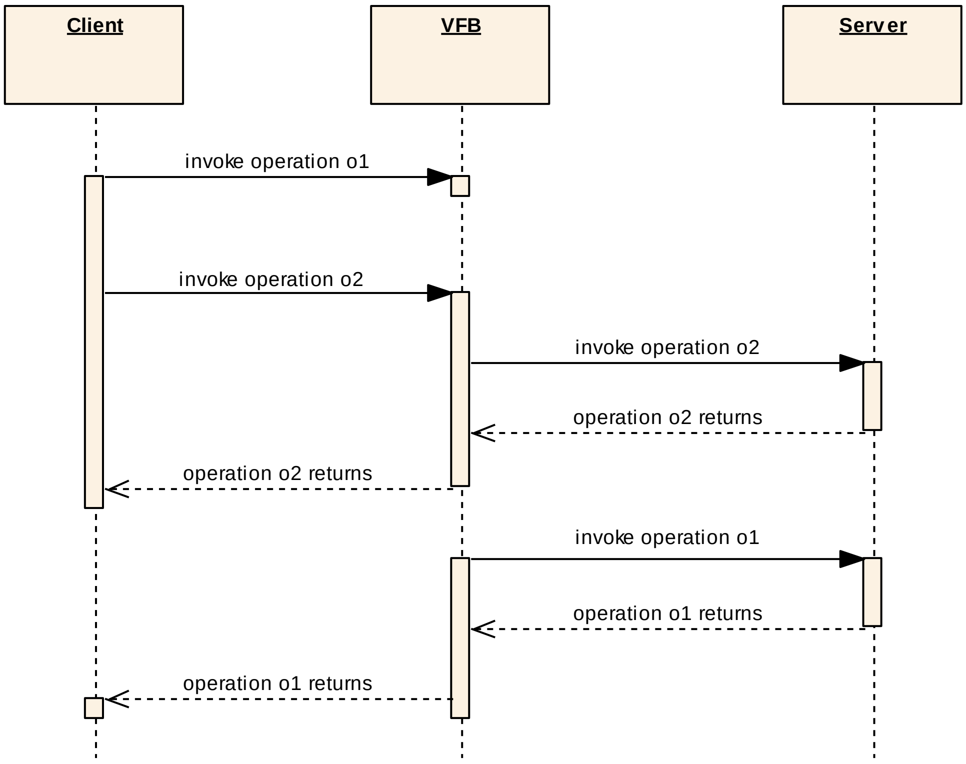


**Figure 4.8: The VFB does not support ordering between different operations**

## 4.5 Remarks regarding the identification of communication partners

One of the main goals of AUTOSAR is the transferability of AUTOSAR software-components and the possibility to integrate the same component in different systems. Therefore, the basic communication mechanisms must not depend on the identity of the communication partners. Which component communicates by which port to which other port of another component is specified by connectors in the VFB View and is not visible to a software-component. If a software-component does need to know the identity of a communication partner for specific communication scenarios the

identification has to be done by the components itself on application level by using the general AUTOSAR communication patterns[17].
By contrast, the unambiguous identification of communication partners, i.e. instances of components and their ports/interface elements, is necessary for the implementation of the RTE and maybe for the basic software[18].

[17] For future extensions like “dynamic components” and “dynamic communication” communication partners have to provide means to be identified on application level.
[18] For example, in client-server communication the result of the invocation of an operation has to be dispatched to the correct client, i.e. the client that invoked the service. Therefore, the identity of the client, i.e. AUTOSAR SW-component and the port, has to be known - at least at runtime - to the RTE and the basic software.

# 5 Timing-model for the VFB (For information only - not part of the Standard)

This section uses a generic timing framework to come to a more precise understanding of the timing-related communication attributes. These concepts are not mature yet and are therefore not part of the standard. They are intended as a basis for future extensions.

## 5.1 Generic timing framework

This section describes the generic framework that is used to describe timing issues. This generic framework will be employed in the following sections to provide a precise definition of the behavior of the VFB.
The concepts are illustrated using UML-diagrams.[19]

### 5.1.1 Event and EventOccurrence

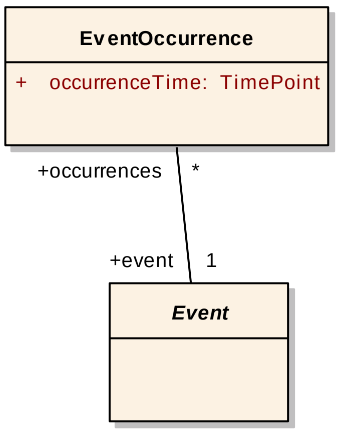


**Figure 5.1: Event and EventOccurence**

Events occur at instantaneous points in time. Therefore, each EventOccurrence has an occurrenceTime, which is the point in time at which the event occurs. An event can occur an arbitrary number of times.
The event itself is usually not defined by its set of occurrences but rather by a description that characterizes the nature of the event.
For example, the “user pressing a key on the keyboard” is an event. This event can occur an arbitrary number of times (namely each time the user presses a key). Each occurrence has a unique occurrenceTime (which is the time the user pressed the key).

[19] Note that because these diagrams are not supposed to represent “templates”, [Template UML Profile and Modeling Guide] does not apply

The distinction between an event and an occurrence of the event is usually obvious from the context. The term event may also be used instead of occurrence of the event in contexts like intervals between events, delay of an event or observing an event.

### 5.1.2 Event Models

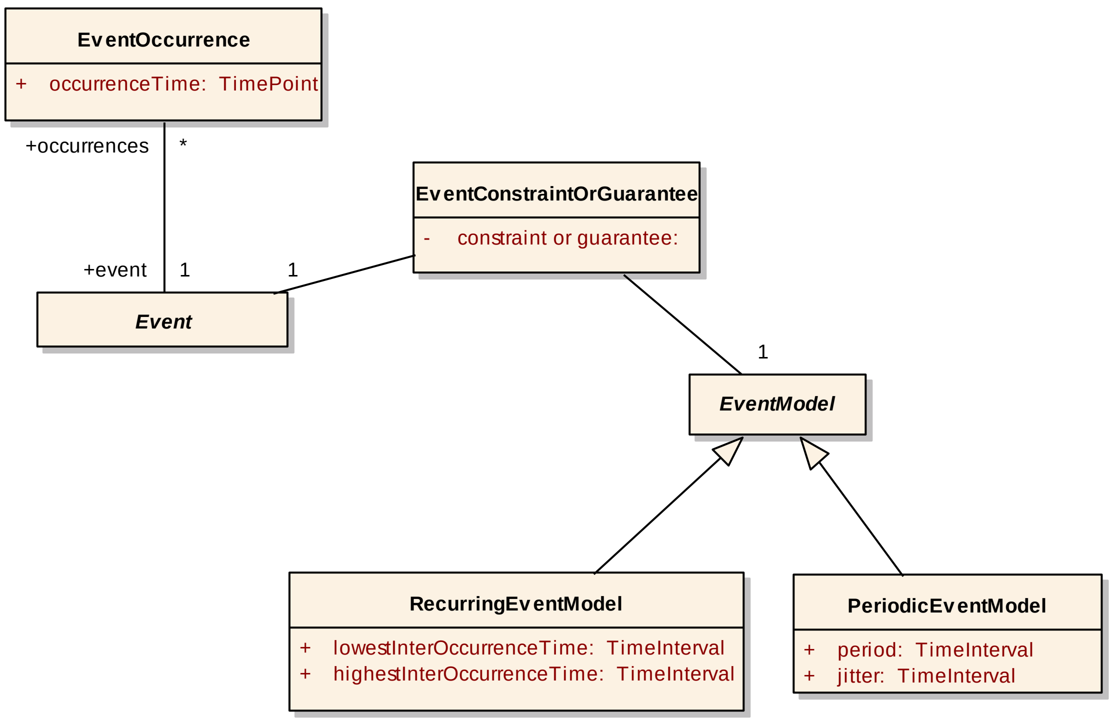

EventOccurrence
+ occurrenceTime: TimePoint
+occurrences
*
+event
1
1
EventConstraintOrGuarantee
- constraint or guarantee:
Event
1
EventModel
RecurringEventModel
+ lowestInterOccurrenceTime: TimeInterval
+ highestInterOccurrenceTime: TimeInterval
PeriodicEventModel
+ period: TimeInterval
+ jitter: TimeInterval

**Figure 5.2: EventModel Overview**

An EventModel can be used to model the occurrenceTimes of an event.
The AUTOSAR timing framework currently defines 2 very simple EventModels.
When the need arises, more sophisticated event models (e.g. statistical models that more precisely describe the distribution of the occurrenceTimes) could be added to the framework.
These event-models can both be used to describe constraints on events or to describe what a module guarantees about the occurrence of events.
For example, a component that can only handle incoming events at a certain rate, can impose the constraint that certain events should be spaced at least 100ms apart (using the RecurringEventModel).
A component containing an internal clock could specify that it guarantees that it produces some data with a certain period and jitter (using the PeriodicEventModel).

#### 5.1.2.1 PeriodicEventModel

The PeriodicEventModel is characterized by the attributes period and jitter, where period must be > 0 and jitter must be >= 0. If an event satisfies the event-model then the difference between the actual time interval between occurrence (n) and occurrence (k) (for any n and k) and their nominal difference (period*(k-n)) must be less than the jitter:

```
| timek - timen - period*(k-n) | <= jitter
```

This basically means that assuming a perfect periodic baseline (of unknown starting point), the differences of the delays of each occurrence compared to this baseline may not be larger than jitter.

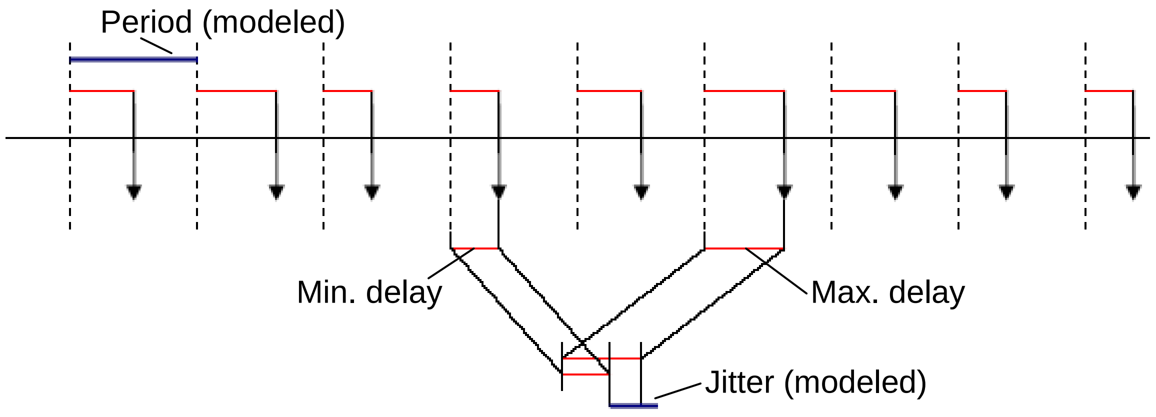


**Figure 5.3: Period and Jitter in the PeriodicEventModel**

Figure 5.3 shows a series of event occurrences where the difference of the minimum and maximum delay (compared to the periodic baseline) is less than the jitter, (as shown on bottom part); therefore the occurrences satisfy a periodic event model with the given period and jitter.

### 5.1.2.2 RecurringEventModel

The RecurringEventModel is characterized by the attributes lowestInterOccurrenceTime and highestInterOccurrenceTime whereby lowestInterOccurrenceTime > 0 and highestInterOccurrenceTime >= lowestInterOccurrenceTime.

This model means that, when an event occurs, the next event occurs in the time-interval:

```
[t_last_occurrence+lowestInterOccurrenceTime,
t_last_occurrence+highestInterOccurrenceTime]
```

This model is typically used to describe sporadic events.

For example, requiring that an event satisfies a RecurringEventModel with lowestInterOccurrenceTime=100ms and highestInterOccurrenceTime=infinity, means that two consecutive events are at least 100ms apart from each other.

### 5.1.3 Timing Chains

Often the description of individual events and their models is not sufficient to characterize the timing behavior of a system. Similarly, the constraints that are used in doing timing analysis are related to the differences in the occurrenceTimes of various related events. A typical requirement could e.g. specify the maximum difference between the times when the front and the rear turn indicators are lit up, or the maximum delay of locking a door after the corresponding button has been pushed.

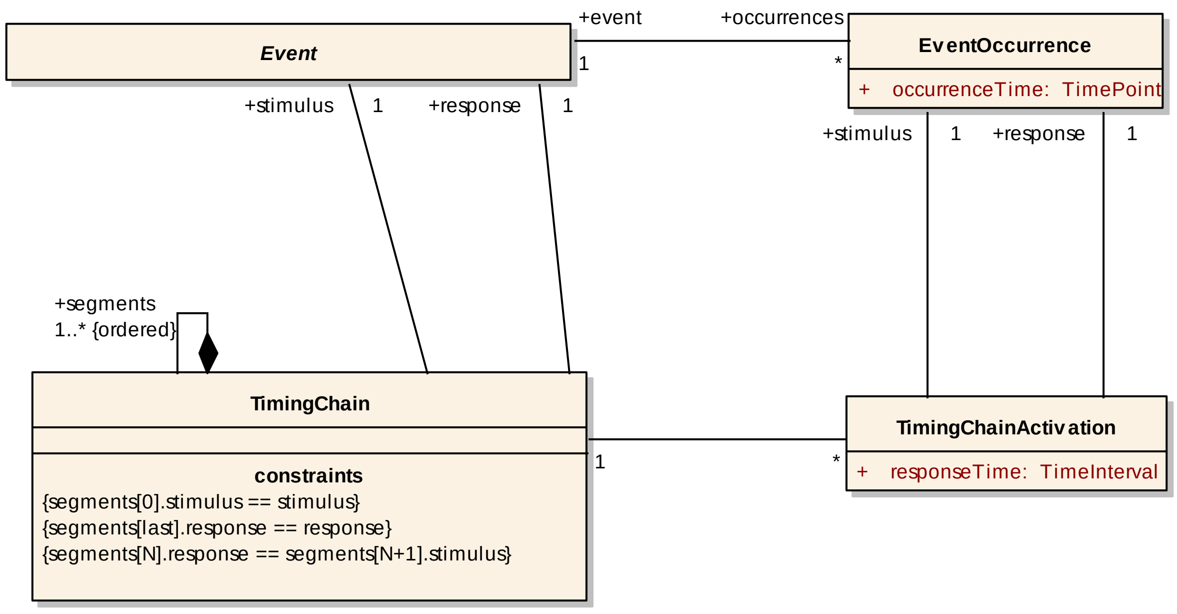

Event
+event
1
+occurrences
*
EventOccurrence
+ occurrenceTime: TimePoint
+stimulus
1
+response
1
+stimulus
1
+response
1
+segments
1..* {ordered}
TimingChain
constraints
{segments[0].stimulus == stimulus}
{segments[last].response == response}
{segments[N].response == segments[N+1].stimulus}
1
*
TimingChainActivation
+ responseTime: TimeInterval

**Figure 5.4: TimingChains**

In order to model related events, the concept of TimingChain is introduced.
A TimingChain is associated with a stimulus-event and a response-event. In the example above, the stimulus would be the event that the “door-lock button has been pressed” and the response would be the event that “all doors are locked”. In case the events carry a value, the timing-chain is typically associated with a specific (potentially complex) relationship between the value of the stimulus and the value associated with the response. For example; if the stimulus is “lock-button changed state” and the response is “doors changed state”, the timing-chain implies that the stimulus with data “locked” will lead to the response with data “locked”. Describing the exact relationship between the values associated with stimuli and responses is not in the scope of AUTOSAR.
A TimingChain can be activated an arbitrary number of times. One TimingChainActivation is then associated with a specific occurrence of the stimulus (the lock-button is pressed at a certain moment in time) and a specific occurrence of the response (all doors are locked, because the button has been pressed). Consequently, each TimingChainActivation can be associated with a responseTime, which is the time-difference between the occurrence of the stimulus and the occurrence of the response.

Note that the presence of a TimingChain in the model (for example describing the chain from the button being pressed to all the doors being locked) does not necessarily mean that each occurrence of the stimulus automatically leads to the activation of the timing chain.

### 5.1.4 Timing Chain Models

TimingChainModels can be used to describe guarantees or constraints on the responseTimes of all TimingChainActivations of a TimingChain and on the relationship between the occurrence of a stimulus and the activation of the TimingChain.

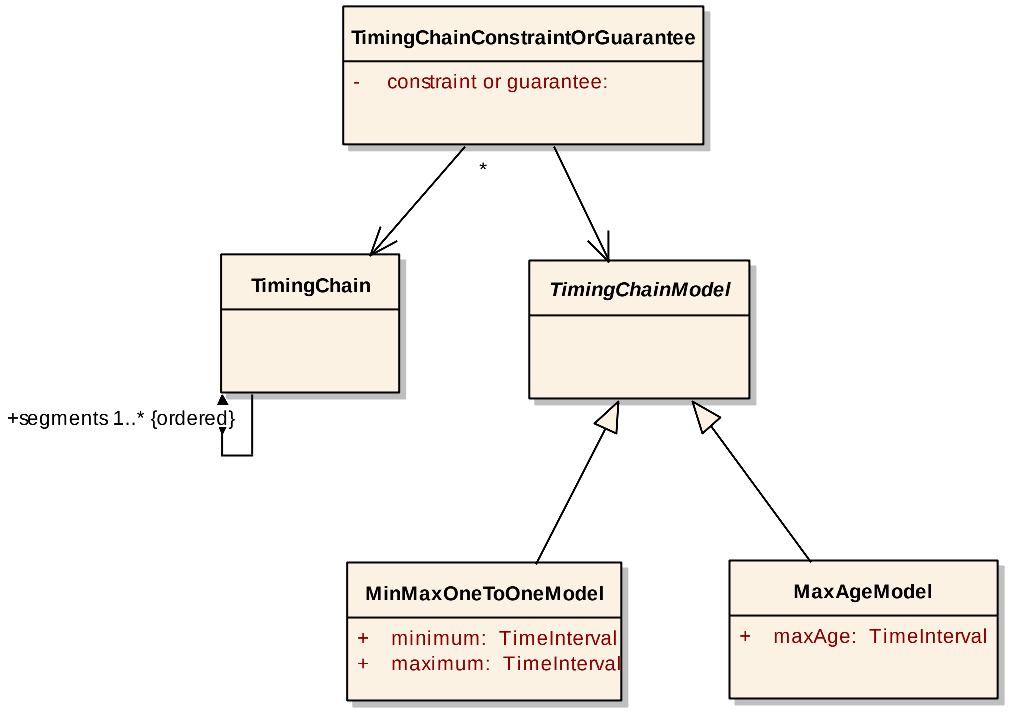


**Figure 5.5: TimingChainModels**

Many sophisticated models could be used to describe responseTimes of timing-chains. The AUTOSAR Timing Model currently defines only two very simple models: the MinMaxOneToOneModel and the MaxAgeModel.

#### 5.1.4.1 MinMaxOneToOneModel

The MinMaxOneToOneModel is characterized by the attributes minimum and maximum, where 0 <= minimum <= maximum.
A TimingChain satisfies the model, when there is a timing-chain activation for EACH occurrence of the stimulus and when the responseTime of each activation satisfies:

```
minimum <= responseTime <= maximum.
```

Often the maximum responseTime is called a “deadline”.

This model can be used to model event-driven systems, where one component is supposed to react in an event-driven way to each stimulus and where the response has to be delivered within a certain defined time-interval after the stimulus.

#### 5.1.4.2 MaxAgeModel

The MaxAgeModel is characterized by one parameter: the maxAge, which is >= 0.
Intuitively, the model is used to describe systems where the events define changes in a state. Often it is not important that a system responds to each change in the value of a state but that the response is new enough with respect to the availability of information on the changes in the state.
A TimingChain satisfies the model when 2 conditions are satisfied:

- for each activation of the timing-chain, the response-time is smaller than the maxAge
- each stimulus S either leads to a response OR
  there is a timing-chain activation with a stimulus S' (occurring AFTER S) and a response R' such that the difference between the occurrenceTime of R' and the occurrenceTime of S is smaller or equal to the maxAge.

## 5.2 Timing aspects of sender-receiver communication

### 5.2.1 Definition of the events

Depending on whether the data-element is associated to a PPort or an RPort, we distinguish between the following events.

DataElementAvailableOnPPortEvent: This event occurs when a value of a specific data-element has been made available by the providing component. On the implementation level, this will typically mean that the value of the data-element is available in the COM-module or in an internal RTE buffer (for intra-ECU communication).

DataElementAvailableOnRPortEvent. This event occurs when a value of a specific data-element is available for the requiring component. On the implementation level, this will typically mean that the value of the data-element is available in the COM-module or in an internal RTE buffer (for intra-ECU communication).

### 5.2.2 The sender-receiver timing chain

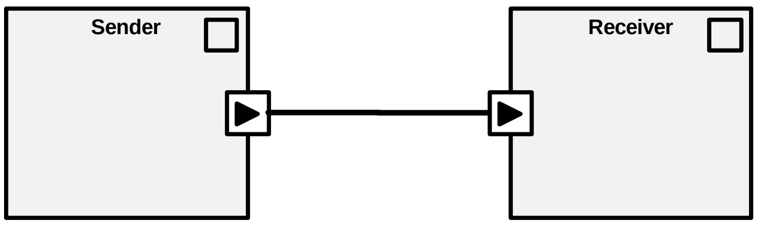


**Figure 5. 6: Example of a simple sender-receiver relationship**

The presence of a connector between the PPort of a component (playing the role of sender) and the RPort of a component (playing the role of receiver) implies the following timing-chain for each data-element in the Sender-Receiver Interface:

Stimulus: a DataElementAvailableOnPPortEvent for the data-element

Response: a DataElementAvailableOnRPortEvent for the data-element

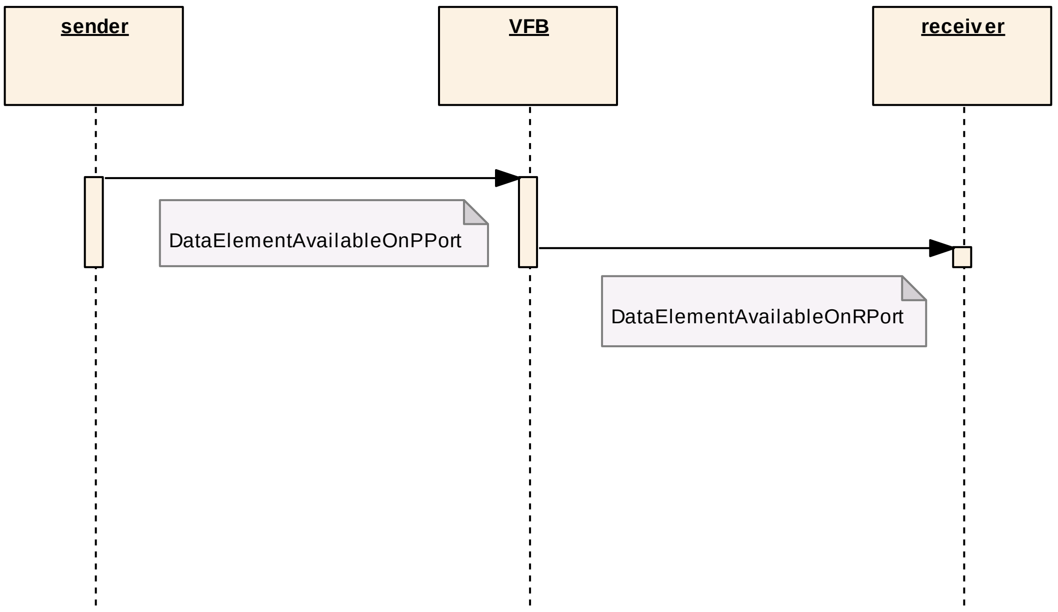


**Figure 5. 7: TimingChain driven Invocation of Operations on the VFB, sender receiver TimingChain**

The timing-chain consists of only one segment which corresponds to the time spent in the infrastructure sending the data-element from the sender to the receiver.

## 5.2.3 Application of the timing framework to sender-receiver

This section explicitly describes the attributes that emerge when applying the generic timing framework on the specific case of sender-receiver communication applied to the transmission of application-events.

### 5.2.3.1 Attributes that are part of the VFB communication attributes

The connector between a PPort and an RPort is responsible for generating a DataElementAvailableOnRPortEvent for each DataElementAvailableOnPPortEvent.
The response-time of the connector is the time between the occurrence of a DataElementAvailableOnPPortEvent and the resulting DataElementAvailableOnRPortEvent.
The appropriate attributes are to set requirements on the behavior of the timing chain using a min-max response-time model. This leads to the attributes:

- MAXIMUM_RESPONSE_TIME_REQUIREMENT: this is the maximal response time allowed by the connector

- MINIMUM_RESPONSE_TIME_REQUIREMENT: this is the minimal response time allowed by the connector

In other words: when a DataElementAvailableOnPPortEvent occurs for a data-element in the PPort, the VFB must ensure that a corresponding DataElementAvailableOnRPortEvent occurs with a response-time between the MINIMUM_RESPONSE_TIME_REQUIREMENT and the MAXIMUM_RESPONSE_TIME_REQUIREMENT.
In many cases, the MINIMUM_RESPONSE_TIME_REQUIREMENT will be 0 (the connector should be “as fast as possible”).

#### 5.2.3.2 Attributes that can be considered for further extensions of the VFB communication attributes

According to the timing-framework it is possible to use an event-model allowing the sender to describe its behavior.
Using the sporadic event-model, this would lead to the following new attributes:

- LOWEST_INTER_OCCURRENCE_TIME_GUARANTEE: The sender guarantees that consecutive generation of events will at least be spaced this time from each other
- HIGHEST_INTER_OCCURRENCE_TIME_GUARANTEE: The sender guarantees that consecutive generation of events will at most be this time from each other

Using the periodic event-model, this would lead to the following new attributes:

- PERIOD_GUARANTEE
- JITTER_GUARANTEE

According to the timing framework the receiver can put requirements on the timing properties of the events arriving at the receiver. In this case, it seems also useful to allow both event-models. This would lead to the following attributes (associated with data-elements in a RPort):

LOWEST_INTER_OCCURRENCE_TIME_REQUIREMENT
HIGHEST_INTER_OCCURRENCE_TIME_REQUIREMENT

OR

PERIOD_REQUIREMENT
JITTER_REQUIREMENT

## 5.3 Timing attributes for client-server communication

### 5.3.1 Definition of the events

To further define the client-server communication semantics, the following events are defined.

**ClientInvocationAvailableOnRPortEvent**: This event occurs when the software component (playing the role of client) has indicated that an operation on an RPort needs to be invoked and has made available values for all outgoing arguments defined for that operation.

**ClientInvocationAvailableOnPPortEvent**: This event occurs when the software component (playing the role of server) has access to the values of all outgoing (incoming from the view of the server) arguments.

**ServerResponseAvailableOnPPortEvent**: This event occurs when the software-component (assuming the role of server) has finished the processing of the service-invocation and has made available values for all returning arguments.

**ServerResponseAvailableOnRPortEvent**: This event occurs when the software-component (assuming the role of client) has the response of the server (including all the values of all returning arguments) available.

### 5.3.2 The client-server timing chain

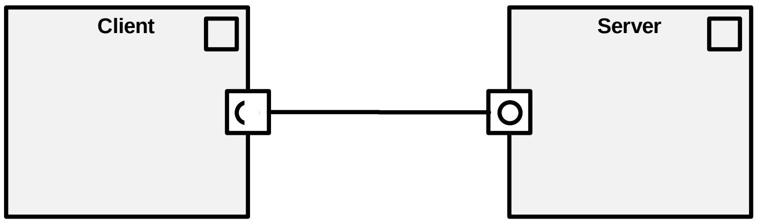


**Figure 5.8: Example of a simple client-server relationship**

The presence of the model shown in the figure above implies the presence of the following timing-chain for each operation in the Client-Server Interface of the connected ports.

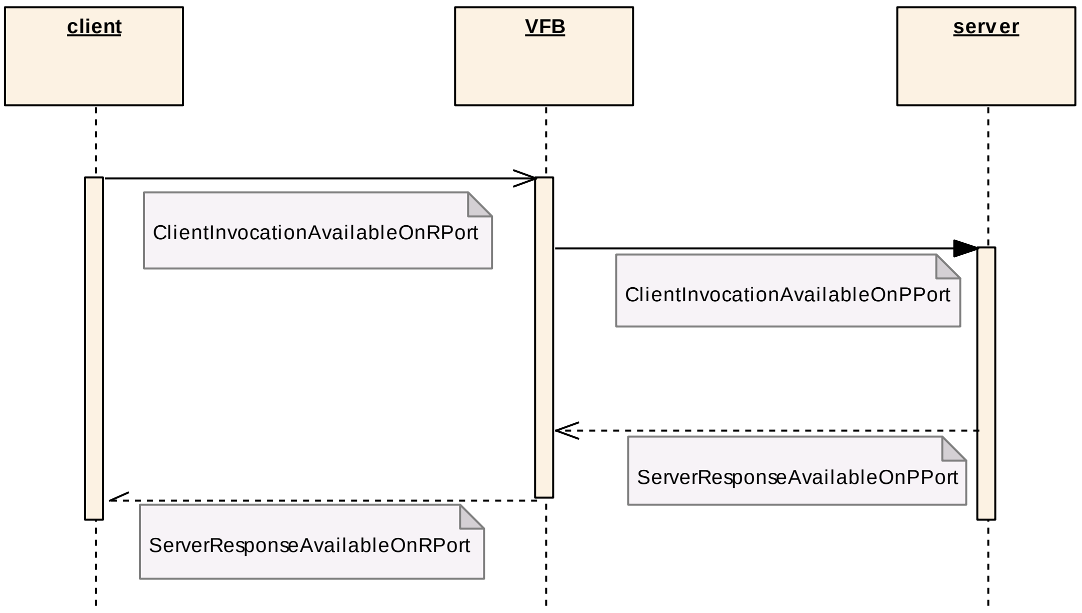

client
VFB
server
ClientInvocationAvailableOnRPort
ClientInvocationAvailableOnPPort
ServerResponseAvailableOnPPort
ServerResponseAvailableOnRPort

**Figure 5.9: TimingChain driven Invocation of Operations on the VFB, client server TimingChain**

This timing chain consists of 4 events:

- Stimulus of the timing chain is a ClientInvocationAvailableOnRPortEvent for a specific operation on the RPort of the client. After this event occurs, the infrastructure can take care of transmitting the service request.
- The 2nd event is a ClientInvocationAvailableOnPPortEvent; after this event occurs the server can start processing the request
- The 3rd event is a ServerResponseAvailableOnPPortEvent
- The final response of the timing chain is the ServerResponseAvailableOnRPortEvent.

The timing chain consists of 3 segments:

- The first segment, between the stimulus and the ClientInvocationAvailableOnPPortEvent corresponds to the VFB transmitting the call from the client to the server
- The second segment corresponds to the server processing the call and providing a response
- The third segment corresponds to the VFB returning the server-response to the client

### 5.3.3 Application of the timing framework to client-server communication

This section describes the communication attributes that are a result of applying the generic timing framework to the specific case of AUTOSAR client-server communication.

#### 5.3.3.1 Attributes that are part of the VFB communication attributes

Based on the timing framework, it is useful to use the min-max model to put a constraint on the response time of the overall timing-chain defined above.
For this purpose, the communication attributes of a client (see Table 4.3) contains the attribute “RESPONSE_TIME_REQUIREMENT”.

#### 5.3.3.2 Attributes that are considered for further extensions of the VFB communication attributes

The software-component playing the role of client, can make guarantees regarding the occurrence pattern of the stimulus “ClientInvocationAvailableOnRPortEvent”. As a sporadic event model is the most appropriate for the invocation of services, the following attributes can be added for each operation on an RPort of a Client:

- LOWEST_INTER_OCCURRENCE_TIME_GUARANTEE: The client guarantees that consecutive invocations of the service will at least be this time apart from each other
- HIGHEST_INTER_OCCURRENCE_TIME_GUARANTEE: The client guarantees that consecutive invocations of the service will at most be this time apart from each other

# 6 Interaction with hardware

## 6.1 Introduction

The goal of this section is to focus on standardized interaction between application software-components and hardware via the Virtual Functional Bus. Hardware interaction means access to the following three kinds of hardware (see also Figure 6.1):

- Microcontroller peripherals
- ECU electronics
- Sensors and Actuators

Actuator and sensor hardware typically needs specialized software to provide an interface towards application software. This interface typically includes a software interface to read sensor values, functions to set an actuator, diagnostic interfaces etc. The integrator needs the flexibility to connect the sensors and actuators of his system to a suitable ECU of his choice.
In some cases, even specialized hardware on the ECU is needed, and an interaction with that hardware is not possible over the standardized basic software. In those cases, complex device drivers may be used to interact with this specific hardware. Complex device drivers are supplier specific.

Figure 6.1 shows the typical conversion process from physical signals to software signals (e.g. car velocity) and back (e.g. car light). This interface architecture is taken because of 2 reasons:

- The best reuse potential (when all other integration requirements like performance requirements are fulfilled):
    - if the µC changes, it is possible to reuse the ECU Abstraction, the sensor-actuator software-component and the application software-component
    - if the ECU changes, it is possible to reuse the sensor-actuator software-component and the application software-component
    - if the sensor or actuator changes, it is still possible to reuse the application software-component
- The various modules can be developed by different experts and/or companies (µC, ECU, Sensor/Actuator, Application)

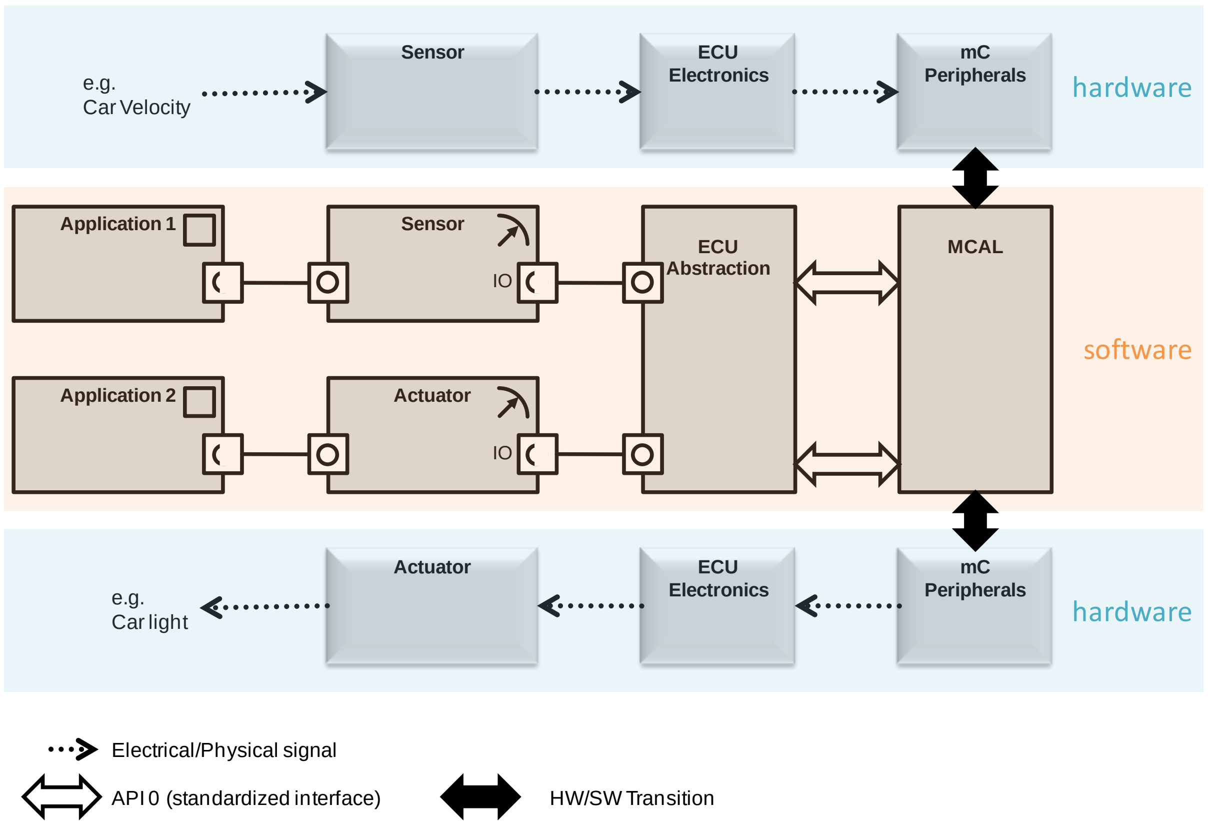


**Figure 6.1: Signal conversions between physical signals and software signals**

## 6.2 Microcontroller Abstraction Layer (MCAL)

Access to the hardware is routed through the Microcontroller Abstraction Layer (MCAL) to avoid direct access to microcontroller registers from higher-level software.
MCAL is a hardware specific layer that ensures a standard interface to the components of the basic software. It manages the microcontroller peripherals and provides the components of the basic software with microcontroller independent values. MCAL implements notification mechanisms to support the distribution of commands, responses and information to different processes.
Among others it can include[20]:

- Digital Input/Output
- Analog/Digital Converter
- Pulse Width (De)Modulator
- EEPROM
- FLASH
- Capture Compare Unit
- Watchdog Timer
- Serial Peripheral Interface
- I²C Bus

[20] Please consult [List of Basic Software Modules] for the actual hardware supported by AUTOSAR.

The MCAL is available on each standard microcontroller.

## 6.3 ECU Abstraction

The ECU Abstraction provides a software interface to the electrical values of any specific ECU in order to decouple higher-level software from all underlying hardware dependencies.
Figure 6.2 shows a typical example for the ECU abstraction. In this case the service "ECU_Set_I" is provided in 3 different ways on the ECU, but the SW-Interface is always the same.

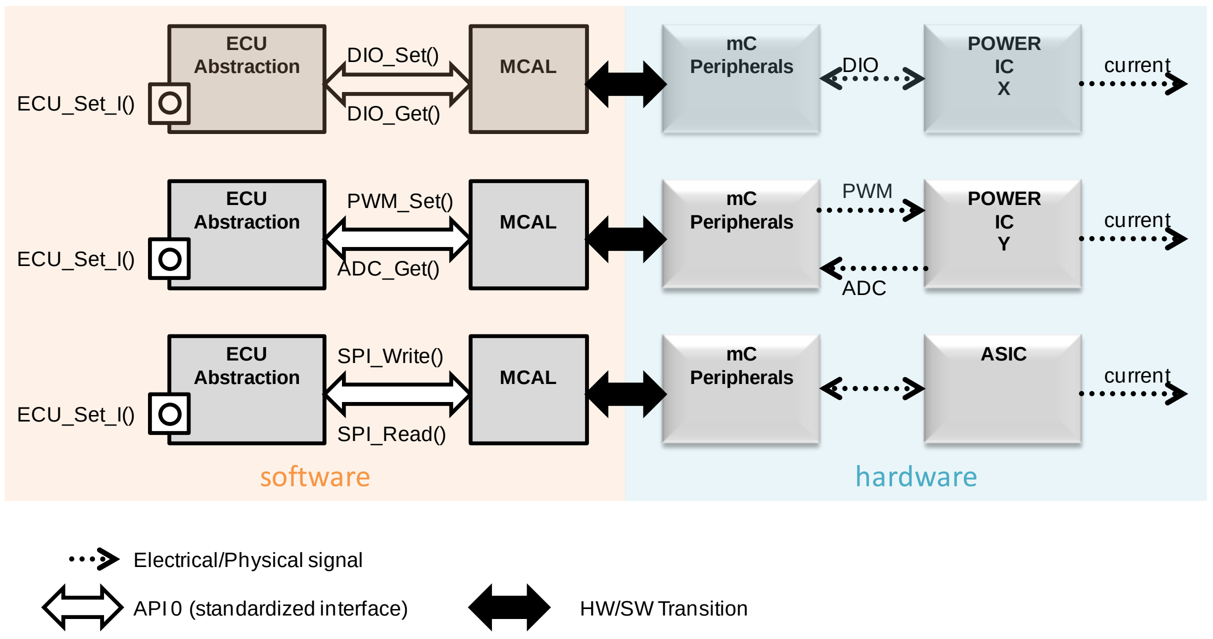


**Figure 6.2: example "ECU_Set_I" for the ECU abstraction**

## 6.4 Sensor-Actuator Software Component

A sensor-actuator software-component is an atomic software-component that makes the functionality of a sensor or actuator usable for other SW-components. That means that the sensor-actuator software-component provides the application software-components an interface for the physical values of the sensors and actuators. A sensor-actuator software-component is written for a concrete sensor or actuator and uses the ECU abstraction interface.

## 6.5 Complex Device Driver Component

The Complex Device Driver (CDD) allows direct access to the hardware in particular for resource critical applications.
The Complex Device Driver is a loosely coupled container, where specific software implementations can be placed. The only requirement to the software parts is that the

interface to the AUTOSAR world has to be implemented according to the AUTOSAR port and interface specifications.
The main task of the complex drivers is to implement complex sensor evaluation and actuator control with direct access to the µC using specific interrupts and/or complex µC peripherals (like PCP, TPU), e.g.

- injection control
- electric valve control
- incremental position detection

Further on the Complex Device Drivers will be used to implement drivers for hardware which is not supported by AUTOSAR.
If for example a new communication system will be introduced in general no AUTOSAR driver will be available controlling the communication controller. To enable the communication via this medium, the driver will be implemented proprietarily inside the Complex Device Drivers. In case of a communication request via that medium the communication services will call the Complex Device Driver instead of the communication hardware abstraction to communicate.
Another example where non-standard drivers are needed is to support ASICs that implement a non-standardized functionality.
Last but not least the Complex Device Drivers are to some extend intended as a migration mechanism. Due to the fact that direct hardware access is possible within the Complex Device Drivers already existing applications can be defined as Complex Device Drivers. If interfaces for extensions are defined according to the AUTOSAR standards new extensions can be implemented according to the AUTOSAR standards, which will not force the OEM or the supplier to reengineer all existing applications.

# 7 AUTOSAR Services

## 7.1 Introduction

This section describes the handling of AUTOSAR services in the VFB view and defines how they can be represented graphically.
AUTOSAR services depict a hybrid concept composed of Basic Software Modules as well as of AUTOSAR Software Components. They provide standardized functionality of the particular ECU infrastructure (AUTOSAR BSW) for Application Software Components mapped onto it.
For the sake of simplicity sometimes the term “service” is used instead of the full term “AUTOSAR service”. However, it has nothing to do with the service part of a client-server interface.

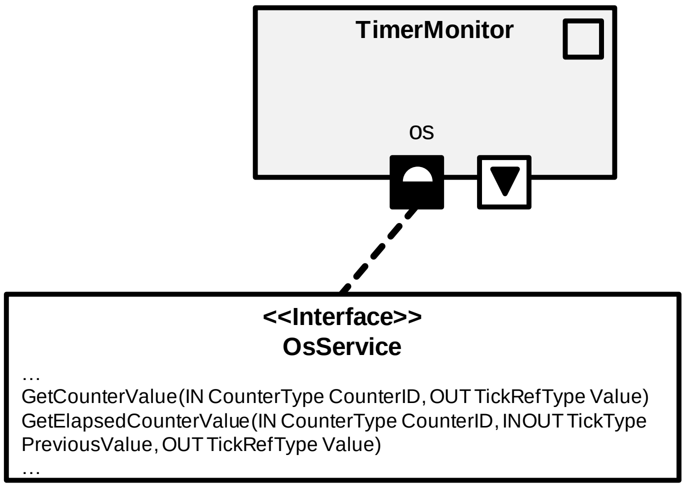

TimerMonitor
os
<<Interface>>
OsService
…
GetCounterValue(IN CounterType CounterID, OUT TickRefType Value)
GetElapsedCounterValue(IN CounterType CounterID, INOUT TickType PreviousValue, OUT TickRefType Value)
…

**Figure 7.1 A software component accesses services of the Os**

Figure 7.1 shows an example for requiring a service: the software component type TimerMonitor has a port typed with the interface OsService. Since this client-server interface contains operations like GetCounterValue or GetElapsedCounterValue, the software component TimerMonitor is able to query the Os about an OsCounter.
Figure 8.3 shows another example: here, the software component has access to the ECU state manager of the ECU Basic Software and its capabilities.

## 7.2 VFB Representation

When it comes to model and configure AUTOSAR services main challenges are:

- the selection of appropriate communication paradigm,
- the fulfillment of prerequisites defined by RTE (see [Specification of RTE Software])
- the platform dependent types
- the configuration

### 7.2.1 Selection of a communication mechanism

In general AUTOSAR services communicate via Standardized AUTOSAR Interfaces. On the VFB they are only visible at the software components requesting the services. The corresponding counterparts in the Basic Software are not visible on the VFB, but inherently present.
Depending on the nature of the service, all kinds of ports are possible:
The most natural way is a service offered to an AUTOSAR component via a provide port typed by a client-server interface: This acts just like a library call returning some data. The corresponding software component would then have a require port like in the example shown in Figure 7.1.
A require port typed by a sender-receiver interface may be used instead, if a service has to be activated but no immediate answer is needed.
A service may also use a require port of typed by a client-server interface in order to communicate with an AUTOSAR component. An example is a state manager, which may need an acknowledgement of an AUTOSAR component before it can change a state.
Instead of the previous case, a service may use the provide port typed by a sender-receiver interface to inform AUTOSAR components about e.g. state changes, if no immediate answer is needed.
In general, the selection of the appropriate communication paradigm is use-case dependent. No general concept except the already defined rules is required. However, note that many services are already predefined by the module specifications of the AUTOSAR Basic Software service layer.
In the VFB view the usage of services by AUTOSAR components is modeled by using a specific graphical notation (see Table 3.2) for ports.
The SWC-Template provides means to attribute the associated interfaces as well as the software components: interfaces mark the attribute isService as true, software components set the attribute ServiceNeeds to an appropriate value.

### 7.2.2 Location of a Service

The examples shown in Figure 7.1 and Figure 8.3 point to a characteristic property of software components accessing specific AUTOSAR services. They can only be integrated onto those ECUs which provide the binding counterparts within the AUTOSAR Basic Software.
This means that the implementation of a service must be located on the same ECU as the AUTOSAR component instance, which is using the service. This is required for good performance and reliability as well as for technical reasons. For example, a timer service is much easier to use locally on the same CPU. For that kind of services we will have instances on different ECUs.

### 7.2.3 Platform dependent types

Many data types within the Basic software are platform dependent to gain efficiency. Especially for IDs holds that the type is dependent on the entities to be handled within a specific ECU, which would definitely restrict the reusability of application software components.

For source code integrated SW-C no problem occurs, because the type will be known at compile time. For SW-C integrated as object code a problem might occur, because the assumed type during compilation of the SW-C might differ from the type assumed by the basic software modules during their compilation.
The solution to this problem is currently that at least parts of SW-C's have to be recompiled after the contract phase although they should be integrated as object code. The integrator in this case has to define the appropriate types and provide the appropriate header file to the suppliers of basic software and application software components.
This results in the restriction that code optimizations within the SW-C and the basic software shall not rely on specific platform dependent types, e.g., the size of data types may vary between different platforms.

#### 7.2.4 Configuration

As most parts of the Basic Software, a service may offer static configuration parameters (i.e. configuration parameters to be defined prior to compile time) in order to be implemented efficiently, e.g. by keeping memory usage low. In many cases these configuration parameters will depend on the number and type of AUTOSAR components by which the service will be used. In these cases at least parts of the software for AUTOSAR services on a specific ECU have to be recompiled at system integration time. Appropriate processes and tools for this have to be specified.
However, this configuration is not part of the VFB view. A good overview of the necessary configuration process needed for AUTOSAR services is given in [Software Component Template].

### 7.3 List of Services

As of AUTOSAR Release 2.1 services of the following BSW modules are available:
- NVRAM Manager – NvM
- Communication Manager – ComM
- Diagnostic Communication Manager – Dcm
- Diagnostic Event Manager – Dem
- Function Inhibition Manager – Fim
- ECU State Manager – EcuM
- Watchdog Manager – WdgM
- Development Error Tracer - DET

# 8 Mode Management

## 8.1 Introduction

Most software components possess specific runnables for initialization, for finalization and for an operational or run mode. The behavior of certain software components might depend in even more complex ways on some system modes. As these components typically do not change their modes themselves, they need to react to mode changes triggered by other components.
Ergo, AUTOSAR needs to support

- The definition of modes
- Communication mechanisms that allow components (including AUTOSAR services) to exchange information about modes and mode-changes
- Scheduling mechanisms that allow components to specify how they behave in different modes

This section briefly describes the generic mechanisms provided by AUTOSAR to support this. These generic mechanisms can then be applied to typical automotive use-cases, such as changes in the ECU's power-state or in the mode of the communication bus.

## 8.2 Defining modes

In AUTOSAR the sender-receiver communication mechanism is used to exchange modes between components. In addition to data-elements, a sender-receiver interface can include so called "ModeDeclarationGroups".
Figure 8.1 shows an example of the definition of the sender-receiver Interface "ECUMCurrentMode" containing a single reference to the ModeDeclarationGroup "ECUMMode".

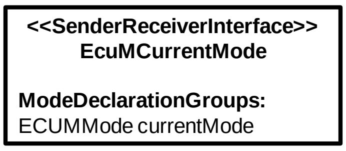


**Figure 8.1: Example of a Sender-Receiver Interface "ECUMCurrentMode" with a single ModeDeclarationGroup**

The ModeDeclarationGroup is a set of ModeDeclarations. Within the definition of the group, one ModeDeclaration describes the initial mode. For example, for the case of the ECU power state, the ModeDeclarationGroup "ECUMMode" could define the group of modes named { STARTUP_SHUTDOWN, RUN, POST_RUN, SLEEP, WAKE_SLEEP }, with STARTUP_SHUTDOWN as the initial mode.
The modes are mutually exclusive: at run-time, there is always one active mode in a ModeDeclarationGroup. The initial mode of a ModeDeclarationGroup is active before any mode switches occurred.

VFB115: There shall be exactly one active mode for each ModeDeclarationGroup in a RPort or PPort of a component

VFB116: At configuration time, the initial mode of each ModeDeclarationGroup in a sender-receiver interface is known

VFB112: At configuration time, it is known which ModeDeclarationGroups a sender-receiver interface contains

VFB114: At configuration time, the modes of each ModeDeclarationGroup in a sender-receiver interface are known

## 8.3 Communicating modes

Modes are transmitted via the sender-receiver mechanism.
There will be software-components that have PPorts typed by interfaces containing mode-declaration groups. The components that provide these interfaces set the current mode within the group and are therefore called "mode-managers".
The counterparts of the "mode-managers" are components whose behavior depends on the current mode. These modules have RPorts typed by the same interface. If the corresponding PPorts and RPorts are connected via a connector, these components are informed about mode-switches and the current mode set by the mode-manager. Figure 8.2 shows an example of this for the case that the mode-manager is an AUTOSAR Service. This figure is an extract out of the example of Figure 3.13.

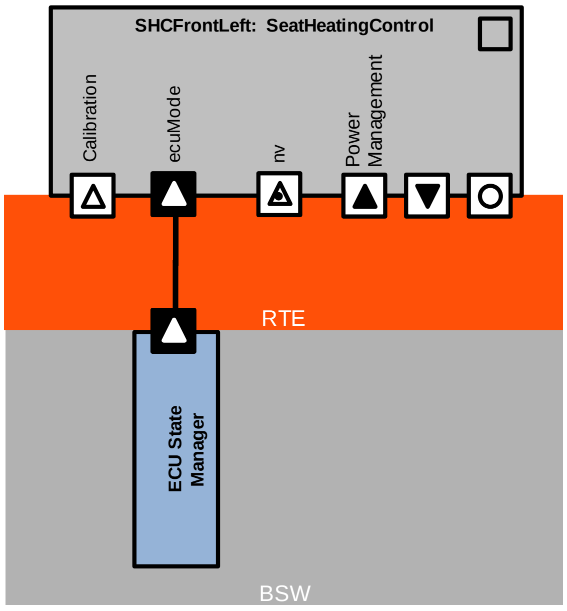


**Figure 8.2: Example of a the communication of a mode from the "ECU State Manager" Service-component to an application software-component**

In the case of an interface containing ModeDeclarationGroups, only 1:n communication (1 sender and n receivers, with n ≥ 0) is possible. The single sender (the mode-manager) owns the current mode of the ModeDeclarationGroup. The receivers are informed of any mode switch of the sender.

## 8.4 Mode-managers: components that control modes

Entering and leaving modes is initiated by a mode manager. A mode manager might for example be the Communication Manager, the ECU State Manager, or an application mode manager. An application mode manager is a software-component that provides the service of switching modes.
Such a mode manager contains a PPort typed by a sender-receiver interface which references the appropriate ModeDeclarationGroup. The state of the mode managers will be sent to other component using sender-receiver communication.

## 8.5 Components that depend on modes

Some software components need to be capable of reacting to state changes issued by mode managers and adapt their behavior to the new situation. Such software-components include an RPort typed by a sender-receiver interface which references the appropriate ModeDeclarationGroup.
Figure 8.3 shows an example whereby the sender-receiver interface "EcuMCurrentMode" is used to type the RPort "ecuMode" of the component "SeatHeatingControl". As the interface contains the ModeDeclarationGroup "ECUMMode", this indicates that the component "SeatHeatingControl" wants to be notified through its port "ecuMode" whenever there is a change in the "ECUMMode" (this could for example be the current mode of the ECU on which the component runs).

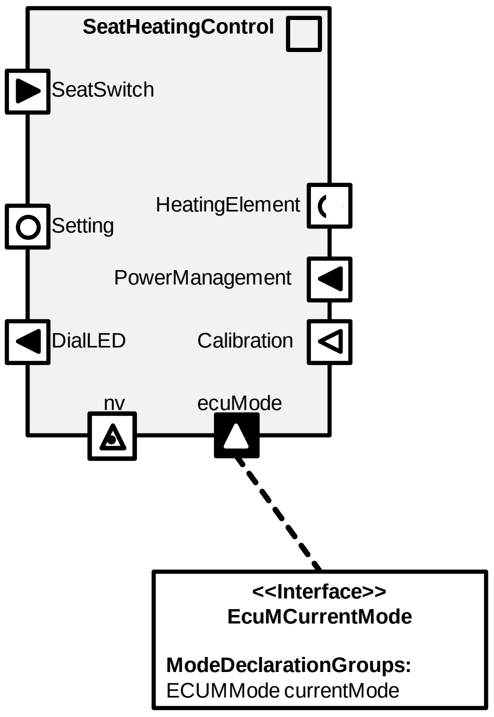


**Figure 8.3: Example showing the use of the Sender-Receiver Interface "ECUMCurrentMode" to type the Port "ecuMode" of the component "SeatHeatingControl"**

VFB117: At configuration time, it must be known which mode switches, the receiver of a ModeDeclarationGroup in a sender-receiver interface wants to be informed of

VFB118: The receiver of a ModeDeclarationGroup in a sender-receiver interface shall be sequentially informed of all configured mode switches received

VFB119: The receiver of a ModeDeclarationGroup sender-receiver interface shall not be informed of a mode switch until the process attached to the last mode switch of this ModeDeclarationGroup finishes

VFB138: The sequence of mode switches as seen by the receiver of the sender-receiver interface for each ModeDeclarationGroup shall be the same as the sequence of corresponding mode switches send by the sender. The order must be the same to guarantee proper mode transitions on the receiver side.

VFB121: Within an individual sender-receiver connector, the VFB guarantees ordering in the mode switches of an individual ModeDeclarationGroup

Since the behavior of an atomic software component is mainly determined by its set of runnables, the component can specify its reaction to mode changes at the level of runnables: the component can specify that certain runnables are called when mode-switches occur or that certain runnables only run in specific modes.

# 9 Measurement and Calibration

In embedded automotive software design, measurement means "monitoring" of ECU internal signals, state variables and intermediate data. It's realized by reading content of memory cells of a running ECU. In AUTOSAR such data is referred to as measurable.
“Calibration” means the manipulation of particular calibration parameters. In general, a calibration parameter characterizes the dynamics of a control algorithm. From a software implementation point of view it is a variable with read-only access during the normal operation of an ECU. Since the calibration parameter can be set by the calibration system, it is possible to manipulate and readjust the determining factors of closed or open control loop algorithms. Thus, calibration plays an important role during the development process until near completion.

## 9.1 Calibration

AUTOSAR provides two mechanisms for calibration:
Port-based calibration: this mechanism is explicitly visible on the VFB and reuses the already described port- and connector-mechanisms
Private calibration parameters: these reside within an atomic software-component.

### 9.1.1 Port-based calibration

This mechanism builds upon the common VFB patterns in the following way:
A component requiring calibration parameters defines an RPort typed by a calibration-parameter interface
The components that contain the actual values of the calibration parameters are called “calibration-parameter components”. In contrast to normal software-components, calibration-parameter components do not possess an internal behavior but are simple containers that provide calibration parameters. They do this through a PPort typed by a compatible calibration-parameter interface.
The fact that a component is calibrated by a specific calibration-parameter component is expressed through a connector between the corresponding ports. The calibration data is made available via the provide port of the calibration-parameter component to a corresponding require port of any software component.
Since in this model the calibration parameters are visible on the virtual bus, calibration-parameter components are the way to express public calibration parameters.
Depending on whether the corresponding components are instantiated or not, several different cases can be distinguished, described in the following.

#### 9.1.1.1 Pure single instantiation

Figure 9.1 shows the simplest case, where a software component has access to a particular set of calibration parameters by ‘receiving’ them via a connection from a providing calibration component.

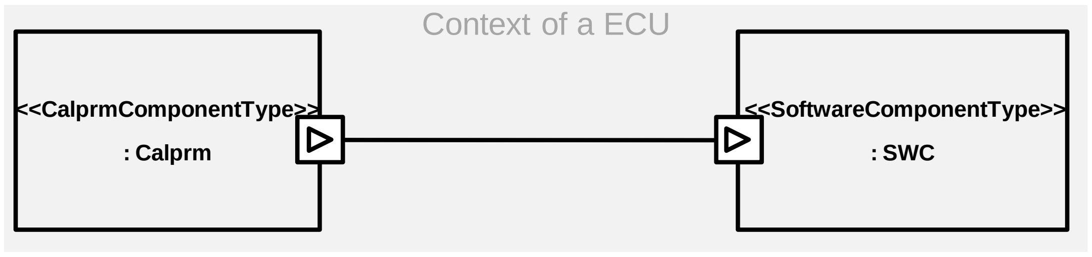


**Figure 9.1 A software component has access to calibration parameter encapsulated in a calibration component**

It should be noted here that the calibration components and software components connected are residing per se on the same ECU. Actually, the calibration components are only representing memory containing the encapsulated calibration parameter.

#### 9.1.1.2 Multiple instantiation of the involved software components

Figure 9.2 and Figure 9.3 depict the case, where several software components (instances) of the same or of different component-type have access to the same set of calibration parameters.

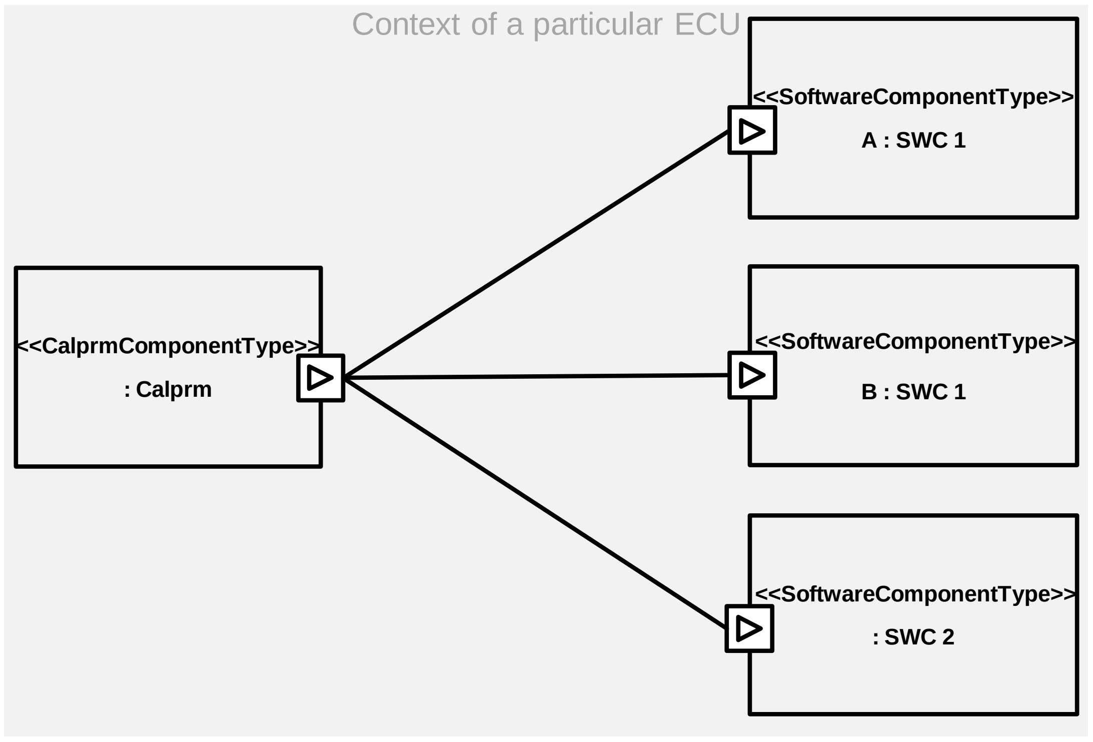

Context of a particular ECU
<<CalprmComponentType>>
: Calprm
<<SoftwareComponentType>>
A : SWC 1
<<SoftwareComponentType>>
B : SWC 1
<<SoftwareComponentType>>
: SWC 2

**Figure 9.2 Two software components of the same type access the same calibration parameter encapsulated in a calibration component**

Since the calibration parameters need to reside on the same ECU as the software component accessing them, the calibration component needs to be duplicated if the different software component instances are mapped onto different ECUs (see Figure 9.3).

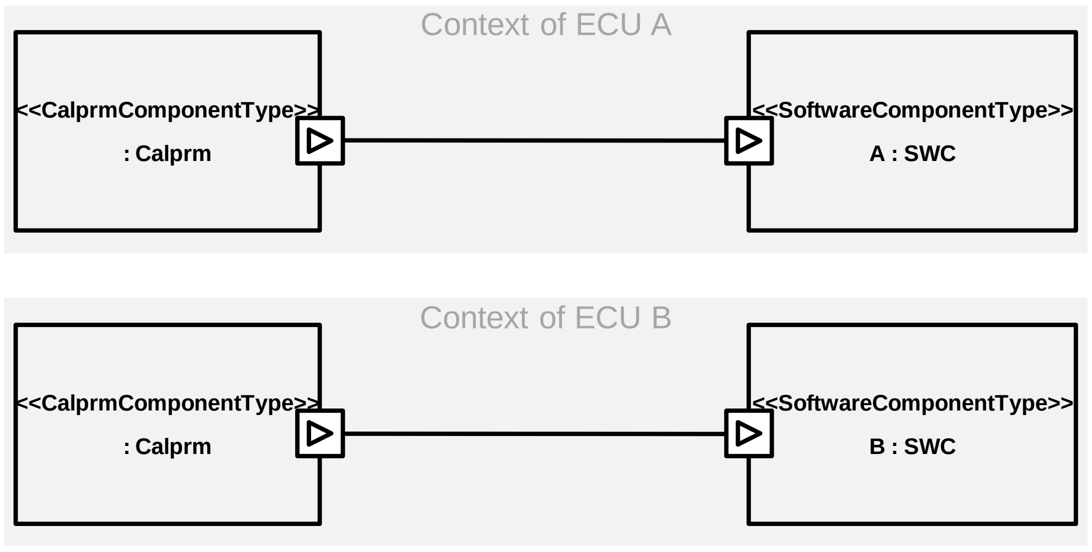


**Figure 9.3 Like in Figure 9.1, but the software components are mapped onto different ECUs**

### 9.1.1.3 Multiple instantiation of the involved calibration components

Figure 9.4 shows a configuration, where different software component instances need to access different sets of the same type of calibration parameter.
Here, it is only required – as explained above – that connected instances of calibration and software components are integrated on the same ECU. Beyond it, the different instances can reside on a single or different ECUs.

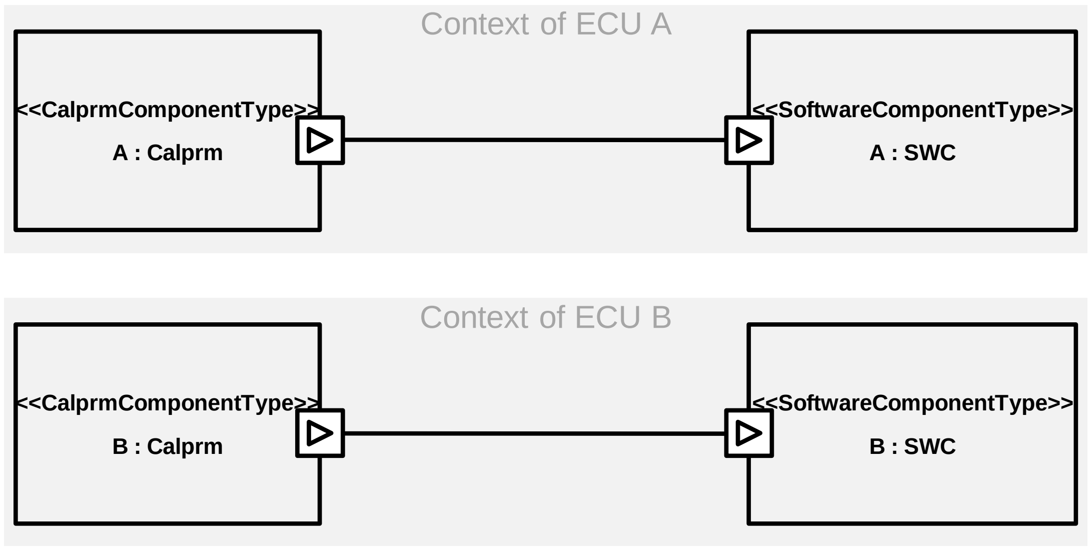


**Figure 9.4 Two software components of the same type have been assigned different instances of the same Calibration Parameter Component Type.**

### 9.1.2 Private calibration

The other calibration mechanism is parameter based and is private and internal to the software component and of type CalprmElementPrototype. CalprmElementPrototypes are used in two different roles, as perInstanceCalprm and as sharedCalprm.
CalprmElementPrototypes used in the role as sharedCalprm operates similar to the scenario described in Section 9.1.1.2, CalprmElementPrototypes used as perInstanceCalprm operates like the scenario given in Section 9.1.1.3.
Calibration parameters of type CalprmElementPrototypes are not visible per se on the virtual functional bus, since it is considered an element associated to an internal behavior of a software component.
Unlike the structure of software components and compositions which is considered to be specified by OEMs, the internal behavior can be defined by the OEMs as well as by their suppliers of particular software components. With this respect the visibility of the calibration parameters modeled as CalprmElementPrototypes is rather a function of time, depending on who and when they are assigned.
Hence, CalprmElementPrototypes are a way for suppliers of software components to express private calibration parameters.

## 9.2 Measurement

In AUTOSAR systems only actual instances of the following prototypes if marked as measurable can be monitored:

- Communication between AUTOSAR SW-Components:
- DataElementPrototypes enclosed in a sender-receiver interface
- Arguments of operationPrototypes enclosed in a client-server interface
- AUTOSAR SW-Component internal
- Content of InterrunnableVariables which are used for communication between Runnables of one AUTOSAR SW-Component.

# 10 Interaction with Non-AUTOSAR-ECUs

## 10.1 Introduction

This section describes the interaction with Non-AUTOSAR-ECUs on VFB level. This kind of interaction is e.g. necessary to provide a migration path.
Non-AUTOSAR-ECUs are:

- ECUs that have not been developed according to AUTOSAR mechanisms. This is useful for e.g.:
    - Integration of an AUTOSAR ECU into an already existing system of ECUs
    - Connect system of AUTOSAR ECUs to already existing system of ECUs
    - Re-use already existing ECU in system of AUTOSAR ECUs
- ECUs that have been developed according to AUTOSAR mechanisms once, but stay unchanged now. This is useful for e.g.:
    - Reuse strategies (taking over of complete unchangeable AUTOSAR (!!!) ECUs)
    - Intelligent ('Smart') Sensors/Actuators with an ECU which do not implement the AUTOSAR VFB / AUTOSAR RTE. This is useful for e.g.:
    - Using Commercial of the shelf LIN nodes.

Interaction of AUTOSAR SW-C with non AUTOSAR software within one ECU is not analyzed in this document.

## 10.2 Problems of interaction

The following problems will arise from the interaction with Non-AUTOSAR-ECUs:

- **Interaction with interfaces of applications on Non-AUTOSAR-ECUs:**
    - Ports/Interfaces have to be mapped to pre-defined communication messages (possible to be routed through gateway)
    - Non-AUTOSAR-SW-Components are currently not modeled at VFB level
        - Unconnected ports of AUTOSAR-SW-Components
        - Hidden communication load
    - Client-Server not supported in old systems.
- **Interaction/support of services implemented on Non-AUTOSAR ECUs**
    - Old services/protocols have to be supported in parallel, to enable interoperability, e.g. Network Management.
    - Additional services supported by communication system (e.g. bus sleep/bus wake-up).
    - LIN nodes inherently are not affected because it is using the master slave paradigm
        - services/protocols have to be managed and implemented in any case by master node (in this case AUTOSAR ECU)
        - Required configuration data available in node capability file (NCF)

- **Problem of support of enhanced services/protocols**
  (e.g. Network Management, Diagnosis (connection to AUTOSAR SW-C), Transport Protocol Layer, ...)

The differentiation, whether the non-AUTOSAR ECU(s) are connected to the same or a different communication system is not relevant for VFB, because no hardware is considered on VFB level. For the same reason gateway configuration is not relevant for the VFB, though it may be a major problem e.g. for the process.

## 10.3 Description of interaction

The modeling of the interaction with non-AUTOSAR-ECUs is done the same for all kinds of non-AUTOSAR-ECUs.

- Non-AUTOSAR ECUs are modeled as separate ECUs with separate AUTOSAR SW-C (with AUTOSAR SW-C Description), which will not be implemented. To enable communication with the non-AUTOSAR ECU the RTE on the AUTOSAR ECU must implement wrapper code for the non-AUTOSAR communication
- Communication messages, configuration and load is defined by System Constraint Template (for LIN Nodes the information contained within the node capability files (NCF) has to be integrated into the System Constraint Template)

The following figure (Figure 10.1: Interaction with non-AUTOSAR ECUs) shall clarify the interaction by giving an example of non-AUTOSAR-ECU(s) interacting with an AUTOSAR ECU. A Port type converter (adapting client server/sender receiver communication) is shown in the example. The port type converter has to be situated on an AUTOSAR-ECU; it doesn’t necessarily need to be on the same ECU the final communication partner is on. As the converter is heir from the class 'AUTOSAR SW-C' it has to be implemented as a separate component. In later solutions it might be part of an automatically generated RTE.
For the sender-receiver communication no adaption is shown. But even when using the same communication paradigm an adaption might be required due to different communication attributes. This would be done the same way like the port type conversion. The adaption has to be implemented as a separate AUTOSAR SW-C; in later solutions it might be done within an automatically generated RTE.
The way between the communication system signals (e.g. signals on CAN) and the RTE layer is the same for AUTOSAR and non-AUTOSAR signals.

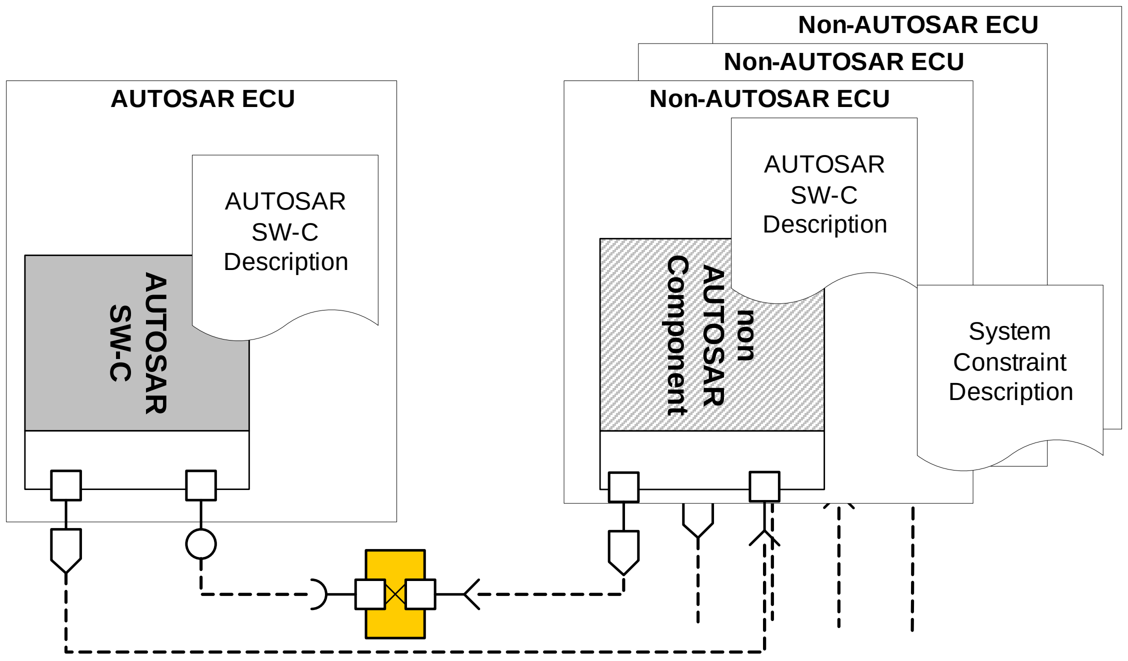


**Figure 10.1: Interaction with non-AUTOSAR ECUs**

The support of enhanced services/protocols (e.g. Network Management, Diagnosis (connection to AUTOSAR SW-C), Transport Protocol Layer, ...) may be handled by Complex Device Drivers or 'special' implementations of the corresponding basic-software module(s).

# 11 References

AUTOSAR Methodology
AUTOSAR_Methodology.pdf

AUTOSAR Glossary
AUTOSAR_ Glossary.pdf

AUTOSAR Technical Overview
AUTOSAR_ TechnicalOverview.pdf

Main Requirements
AUTOSAR_MainRequirements.pdf

List of Basic Software Modules
AUTOSAR_BasicSoftwareModules.pdf

Layered Software Architecture
AUTOSAR_LayeredSoftwareArchitecture.pdf

Software Component Template
AUTOSAR_SoftwareComponentTemplate.pdf

Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

Specification of Graphical Notation
AUTOSAR_GraphicalNotation.pdf

Specification of RTE Software
AUTOSAR_SWS_RTE.pdf

Template UML Profile and Modeling Guide
AUTOSAR_TemplateModelingGuide.pdf