Document Title	# Virtual Functional Bus
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	056
Document Classification	Auxiliary

Document Version	2.2.0
Document Status	Final
Part of Release	4.0
Revision	3

Document Change History

Date	Version	Changed by	Change Description
13.10.2011	2.2.0	AUTOSAR Administration	• Enhanced graphical notation (NV data interface support) • Introduction of a mixed conversion block • Clarification of the use of AUTOSAR services within compositions
11.10.2010	2.1.0	AUTOSAR Administration	• Improved description of port compatibility and data conversion scaling • Improved consistency to other AUTOSAR specifications • Fixed outdated graphical notation in images • Reformulated description of timing extension
30.11.2009	2.0.0	AUTOSAR Administration	• Introduction of new concepts (Variant Handling, Integrity and scaling at port, Mode Management, Triggers, Access to NVM, access to parameters and calibrations) • Synchronization with the current AUTOSAR Meta-Model (new interfaces and SwComponentTypes) • Timing extension moved to the AUTOSAR_TPS_TimingExtensions document • Legal disclaimer revised
23.06.2008	1.0.1	AUTOSAR Administration	Legal disclaimer revised
14.11.2007	1.0.0	AUTOSAR Administration	Initial Release

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.
Useful pre-reads are the “Main Requirements” [3]. Documents that can be consulted in parallel to this document include the “Methodology” [1] and the “Glossary” [2].

1.3 Relationship to other AUTOSAR specifications

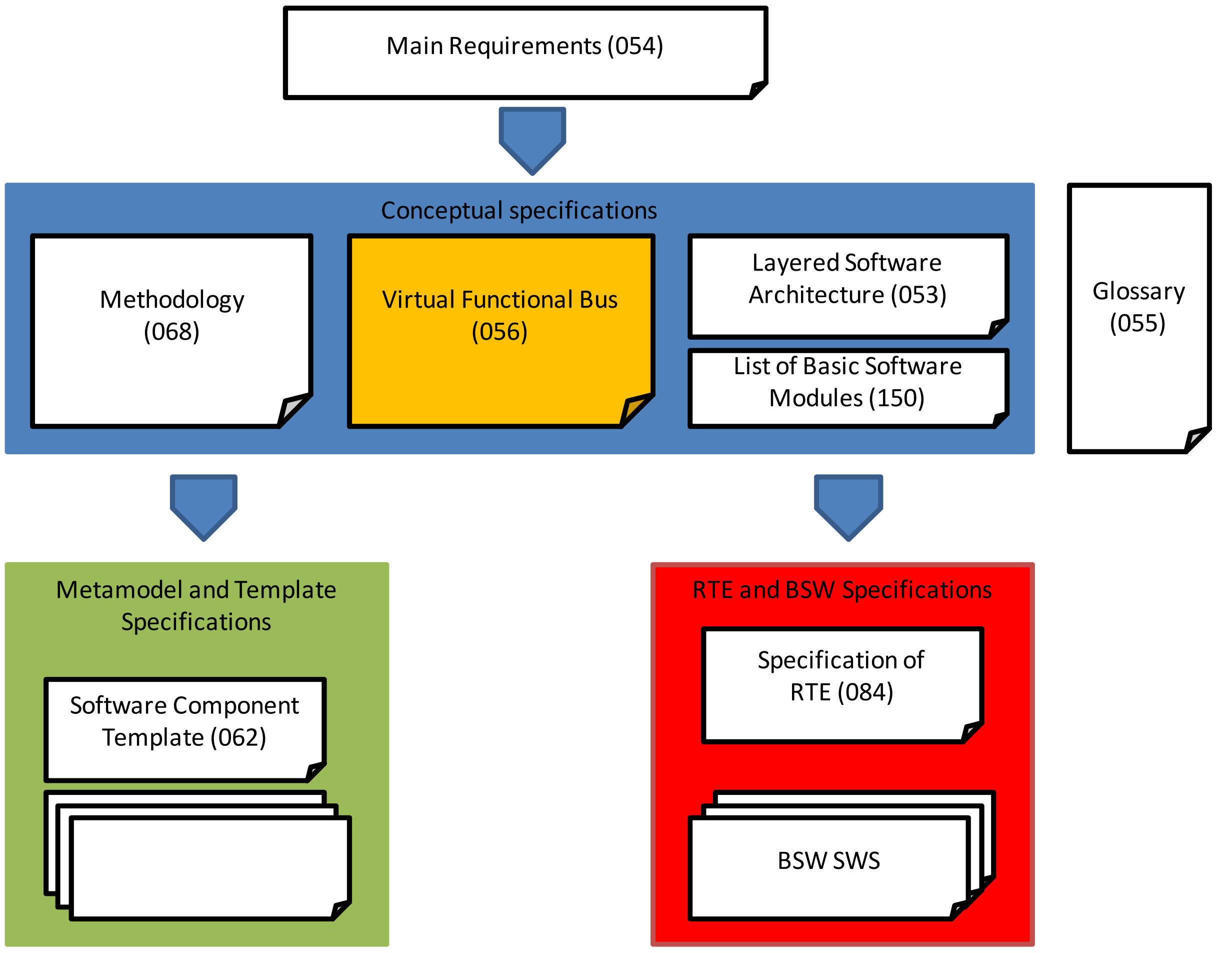

Main Requirements (054)
Conceptual specifications
Methodology (068)
Virtual Functional Bus (056)
Layered Software Architecture (053)
List of Basic Software Modules (150)
Glossary (055)
Metamodel and Template Specifications
Software Component Template (062)
RTE and BSW Specifications
Specification of RTE (084)
BSW SWS

Figure 1.1: Relationship of the Specification of the "Virtual Functional Bus" to other AUTOSAR specifications[1]

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 "Methodology" [1] describes the method that is used when building systems with AUTOSAR
- the specification of the "Virtual Functional Bus"
- the "Layered Software Architecture" [5]
- and the "List of Basic Software Modules" [4]

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

[1] The numbers in brackets refer to the Document Identification Number of the specification.

- The specifications defining the AUTOSAR meta-model and templates: In this group the “Software Component Template” [6] 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” [7] 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.

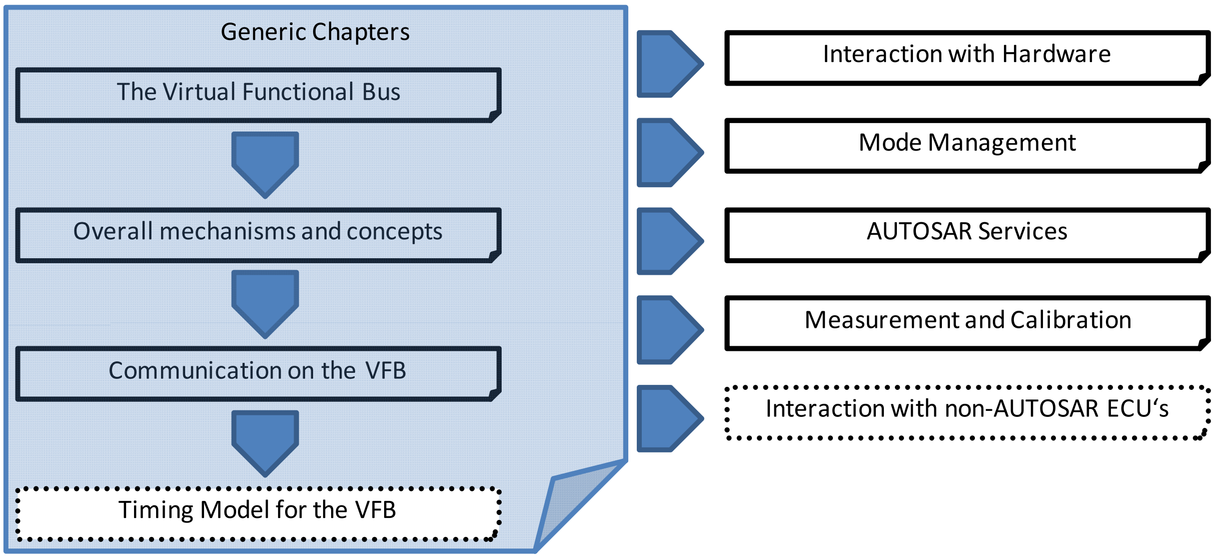

Generic Chapters
The Virtual Functional Bus
Overall mechanisms and concepts
Communication on the VFB
Timing Model for the VFB
Interaction with Hardware
Mode Management
AUTOSAR Services
Measurement and Calibration
Interaction with non-AUTOSAR ECU‘s

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 “Methodology” specification [1]. Figure 2.2 illustrates the “Configure System” activity out of the methodology (top-left), which focuses on the VFB.

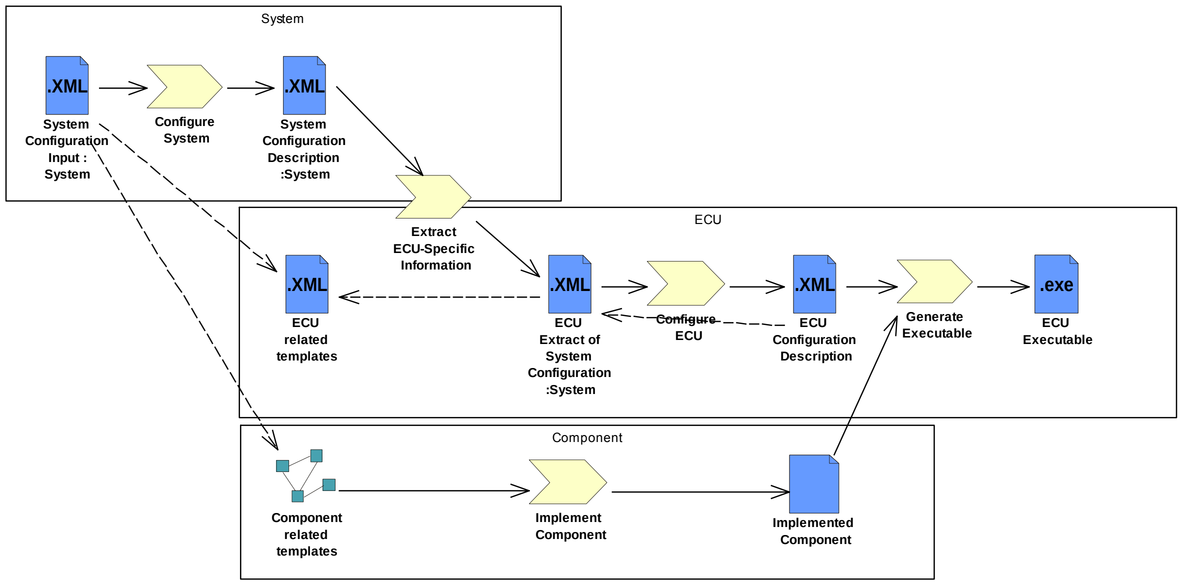

System
.XML
System Configuration Input : System
Configure System
.XML
System Configuration Description :System
Extract ECU-Specific Information
ECU
.XML
ECU related templates
.XML
ECU Extract of System Configuration :System
Configure ECU
.XML
ECU Configuration Description
Generate Executable
.exe
ECU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

Figure 2.1: Overview of the AUTOSAR Methodology [1]

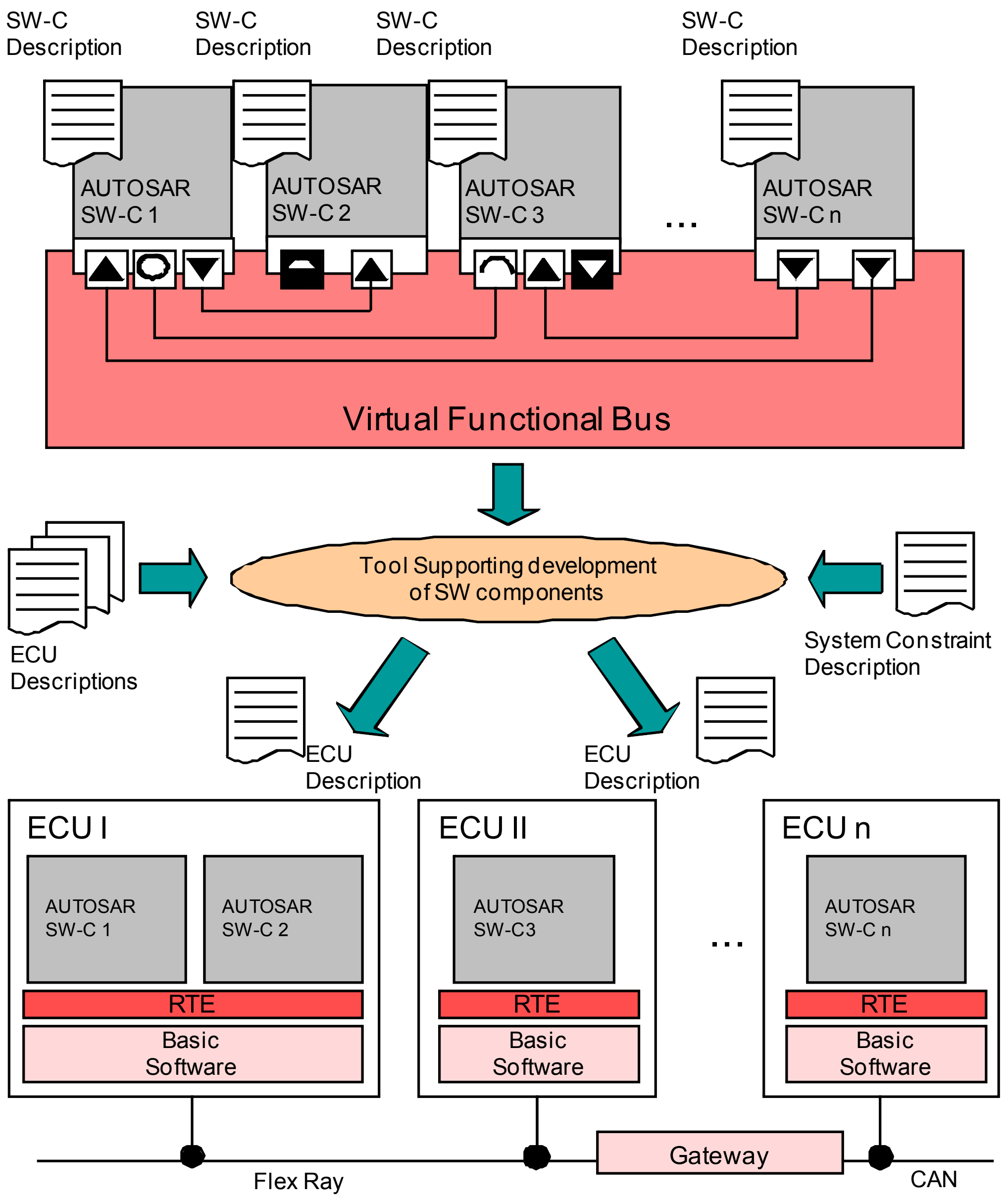


Figure 2.2: Detailed view on the activity “Configure System”

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 individual components and between the components and the Basic Software (BSW) [5][4] is called the Run-Time Environment (RTE) [7]
A component encapsulates complete or partial automotive functionality. Components consist of an implementation and of an associated formal software-component description (defined in the “Software Component Template” specification [6]). 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 (see also AUROSAR "Main Requirements" [3]).
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 10)

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-type requires the following information as input:
- 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.

It controls
- the DialLED that is 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”).

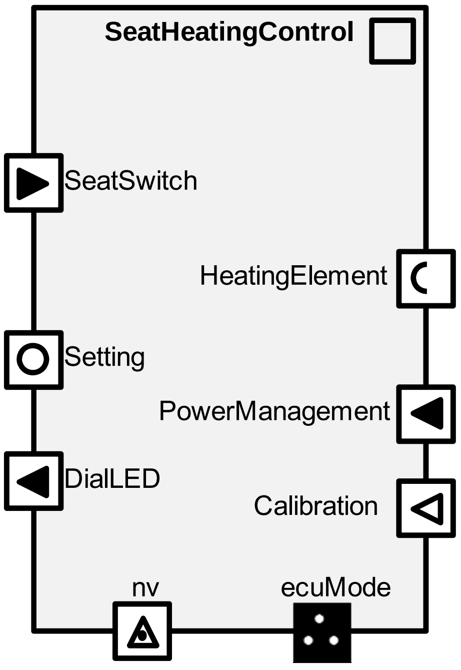


Figure 3.1: Example of the definition of the component-type “SeatHeatingControl” with eight ports

Figure 3.2 shows an example of the definition of a sensor-actuator component[2] 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”).

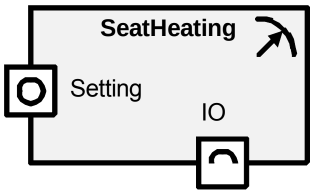


Figure 3.2: Example of the definition of a component-type “SeatHeating” with two ports

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[3] 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).

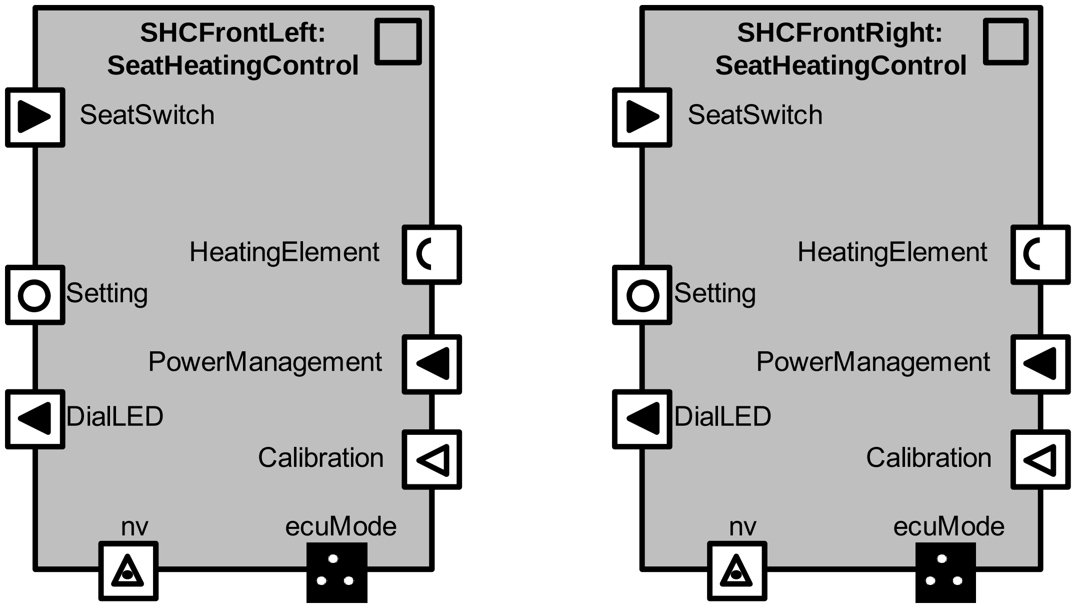


Figure 3.3: Example showing the multiple instantiation of the component “SeatHeatingControl” as “SHCFrontLeft” and “SHCFrontRight”

[2] Chapter 6, Interaction with hardware, defines the exact purpose of the “sensor-actuator” components
[3] Dynamic instantiation at runtime is not in scope of the present release of AUTOSAR.

[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」()

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 port-interfaces supported by AUTOSAR.

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[4]. A mode manager can notify mode switches to one or several receivers	this section and Section 4.3
Parameter Interface	A parameter interface allows software components access to either constant data, fixed data or calibration data. It should be noted that depending on the type of access (i.e. fixed, const or standard respectively) that compatibility rules apply. For example a parameter interface which uses a fixed implementation policy will not be allowed to connect to a port of a Parameter SW Component if the provider uses a variable data implementation (i.e. standard). The reason is plain and simple; The application will use a #define (pre-compile value optimization) and so will not take actual values from the Parameter SW component at runtime.	Chapter 10
Non volatile Data Interface	Provide element level access (read only or read/write) to non volatile data as opposed to NV block access.	Section 4.3
Trigger Interface	The trigger interface allows software components to trigger the execution of other software components. The purpose of the trigger interface is to allow for fast response times with regards to the occurrence of a trigger which might occur sporadic or at a variable cycle time. Example: triggering based on the crank shaft and cam shaft position.	Section 3.8

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

Mode Switch Interface	The mode switch interface is used to notify a software component of a mode. The mode manager provides modes that can be used by mode users to adjust the behavior according to modes or synchronize activities to mode switches.	Section 8

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".

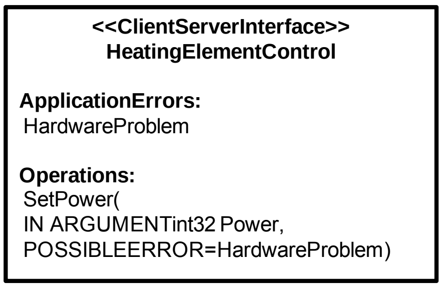


Figure 3.4: Example of a client-server interface "HeatingElementControl" with a single operation

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".

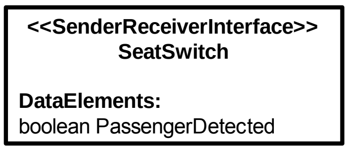


Figure 3.5: Example of a Sender-Receiver Interface "SeatSwitch" with a single data-element

[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

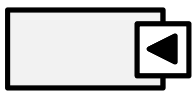
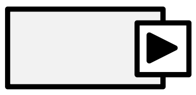
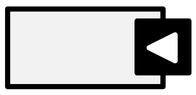
As defined before, the ports of a component are the interaction points between components.
A port of a component is either a “PPort” or an “RPort”. A “PPort” provides the elements defined in a port-interface. An “RPort” requires the elements defined in a port-interface. A port is thus typed by exactly one port-interface[5].

3.3.1 Port Types

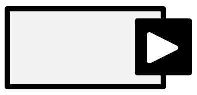
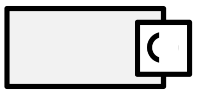
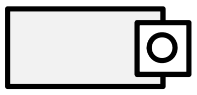
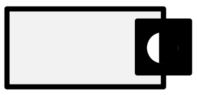
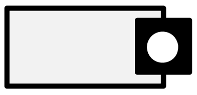
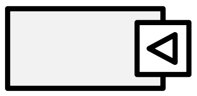
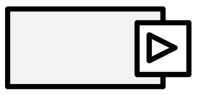
A single port-interface can type several different ports.

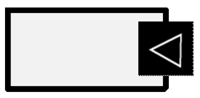
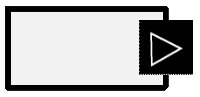
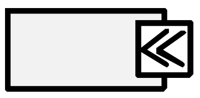
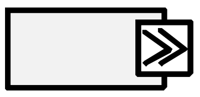
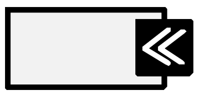
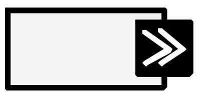
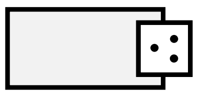
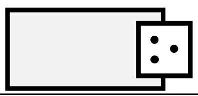
[VFB007]「 At configuration time, it is known whether a component’s port is a PPort or an RPort」()

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
RPort	sender-receiver	No	 The component reads/consumes values of data-elements
PPort	sender-receiver	No	 The component provides values of data-elements
RPort	sender-receiver	Yes	 The component reads/consumes values of data-elements from an

[5] 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.

			AUTOSAR service
PPort	sender-receiver	Yes	 The component provides values of data-elements to an AUTOSAR service
RPort	client-server	No	 The component requires (=uses or invokes) the operations defined in the interface
PPort	client-server	No	 The component provides (=implements) the operations defined in the interface
RPort	client-server	Yes	 The component requires (=uses or invokes) the operations defined in the interface from an AUTOSAR service
PPort	client-server	Yes	 The component provides (=implements) the operations defined in the interface to an AUTOSAR service
RPort	parameter (this includes requiring calibration data)	No	 The component requires parameter data (either fixed, const or variable)
PPort	parameter (this includes providing calibration data)	No	

			The component provides parameter data (either fixed, const or variable)
RPort	parameter (this includes requiring calibration data)	Yes	 The component requires parameter data (either fixed, const or variable) from an AUTOSAR service
PPort	parameter (this includes providing calibration data)	Yes	 The component provides parameter data (either fixed, const or variable) to an AUTOSAR service
RPort	Trigger	No	 Component with a trigger sink
PPort	Trigger	No	 Component with a trigger source
RPort	Trigger	Yes	 Component with a trigger sink from an AUTOSAR service
PPort	Trigger	Yes	 Component with a trigger source to an AUTOSAR service
RPort	mode switch	No	 Component with a mode switch user
PPort	mode switch	No	

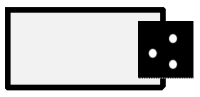
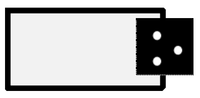
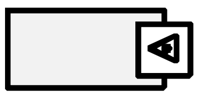
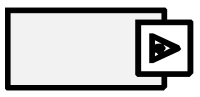
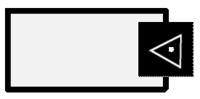
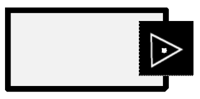
			Component with a mode switch manager
RPort	mode switch	Yes	 Component with a mode switch user with an AUTOSAR service
PPort	mode switch	Yes	 Component with a mode switch manager with an AUTOSAR service
RPort	NV data	No	 The component requires access to non volatile data provided by an NV Block Component
PPort	NV data	No	 The NV Block Component provides access to non volatile data
RPort	NV data	Yes	 The component requires access to non volatile data provided by an AUTOSAR service
PPort	NV data	Yes	 The component provides access to non volatile data to 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”.

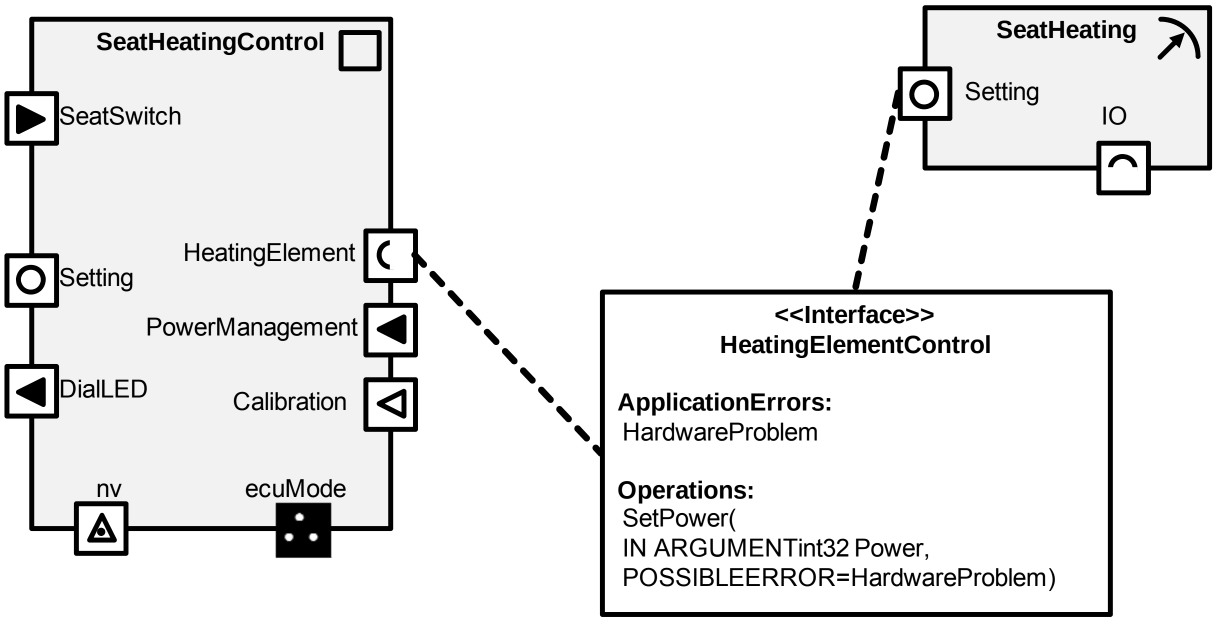


Figure 3.6: Example showing the use of the Client-Server Interface “HeatingElementControl” to type the Port ”HeatingElement” of the component “SeatHeatingControl” and the port “Setting” of the component “SeatHeating”

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”.

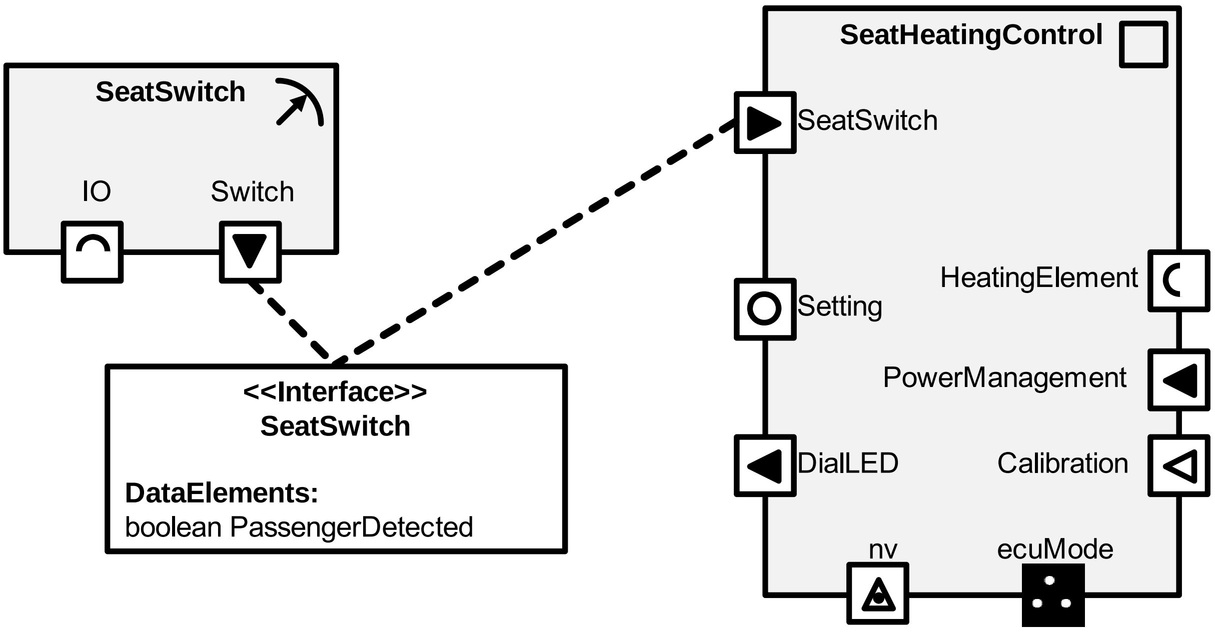


Figure 3.7: Example showing the use of the Sender-Receiver Interface "SeatSwitch" to type the Port "SeatSwitch" of the components "SeatHeatingControl" and the port "Switch" of the component "SeatSwitch"

3.3.2 Port Compatibility

A receiver port can only be connected to a compatible provider port. Table 3.3 gives an overview over the compatibility of ports. The following comments describe some basic compatibility rules. Please note that this overview only contains some basic rules. A more comprehensive and detailed description is given in the "Software Component Template" [6].

(1) For each element in the interface of the require port there must be a compatible element in the interface of the provide port. The mapping is realized implicitly via the shortname of the element or explicitly via explicit mappings (see section 3.9.1).
(2) For mode switch ports all elements of the interface of the provide port must have a corresponding element in the interface of the require port.
(3) Require and provide port are both service ports or are both not service ports.
(4) For connecting ports with Sender Receiver Interface, Parameter Interface or Non Volatile Data Interface, corresponding elements must have compatible implementation policies (see "Software Component Template" [6]).

For example, a Require Port that expects a fixed parameter can only be connected to a Port that provides a fixed Parameter. This is because this fixed data may be used in a compilation directive like #if and only macro #define (fixed data) can be compiled in this case.

Kind of port		Require Port					
	Kind of interface	Sender Receiver	Parameter	Non Volatile Data	Client Server	Trigger	Mode Switch
Provide Port	Sender Receiver	yes (1,3,4)	no	yes (1,3,4)	no	no	no
	Parameter	yes (1,3,4)	yes (1,3,4)	yes (1,3,4)	no	no	no
	Non Volatile Data	yes (1,3,4)	no	yes (1,3,4)	no	no	no
	Client Server	no	no	no	yes (1,3)	no	no
	Trigger	no	no	no	no	yes (1,3)	no
	Mode Switch	no	no	no	no	No	yes (1,2,3)

Table 3.3: Compatibility of kinds of ports
(numbers in this table correspond to the compatibility rules described before)

3.3.3 Data Type Policies

Data elements on a port are typed properly as part of the port interface description of a SWC. However it should be noted though that the data type of elements to be communicated between two ports can be overridden by the integrator by overriding the data type using a data type policy which allows for reducing the number of bits to be transmitted over a physical network. The data type has to be compatible and usually result in loss of precision and introduce quantization artifacts.

3.4 Connectors

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

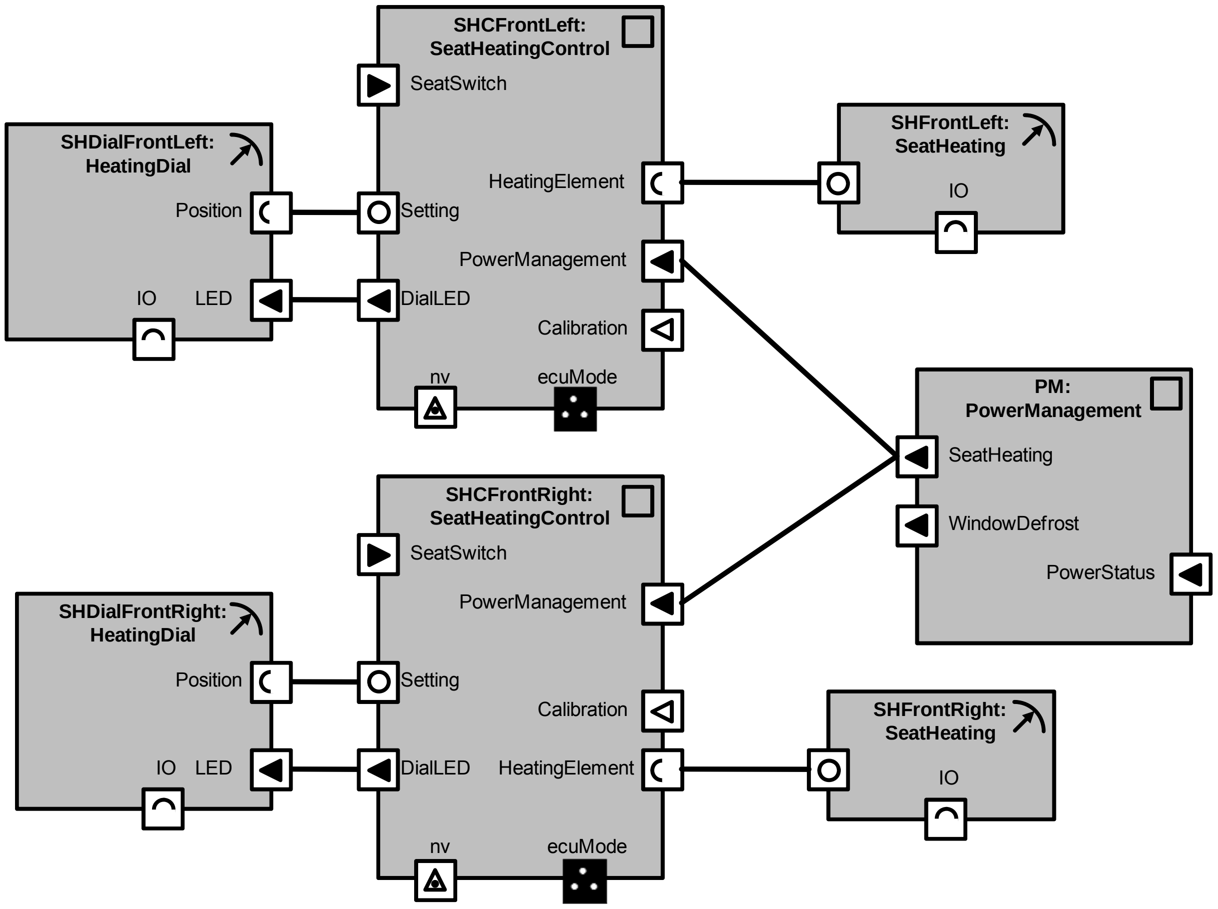


Figure 3.8: Example of the use of eight assembly-connectors to connect the ports of seven components

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.[6]」()

[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[7].」()

3.4.1 Unconnected Ports

The occurrence of an unconnected port is not per se a design mistake. It can be valid when an application provider for the data element is absent and the default init value is good enough to operate with or it could be that an end point was removed from the system because it is subjected to variability (See section Variant Handling).

3.4.1.1 Unconnected Sender/Receiver Ports

If a PPort of a sender receiver communication is unconnected then the data being published by the provider will not appear on the VFB and as such will not be accessible by any other software component.
If an RPort of a synchronous sender receiver communication is unconnected then the RPort shall provide the initial value for the data that is being accessed. For asynchronous communication the client must behave as if the provider times out.

3.4.1.2 Unconnected Client/Server Ports

If a PPort of a client server communication is not connected the server will not receive any requests.

[6] 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.
[7] The exact meaning of “compatibility” is defined in the Software Component Template [6].

If an RPort of a client server communication is unconnected then the VFB behavior shall be as if the server did not respond in time and the client experiences a TIME_OUT.

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.

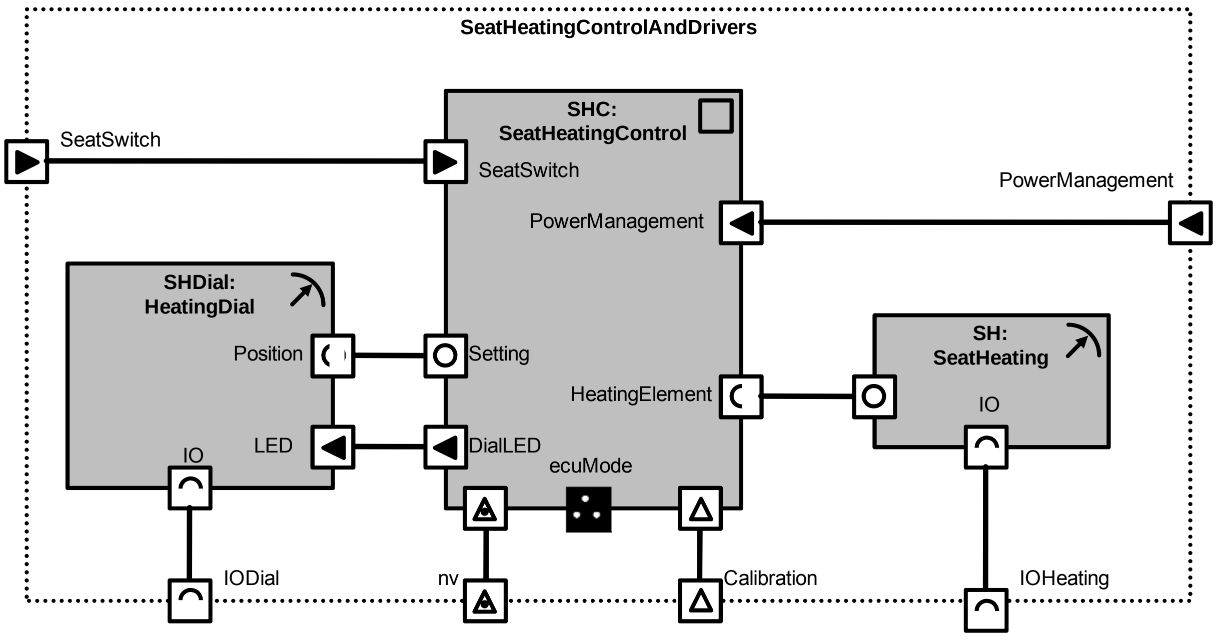


Figure 3.9: Example of the definition of the Composition “SeatHeatingControlAndDrivers”

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.

When designing a composition, service ports have to be specially handled. The configuration of AUTOSAR services takes place in the ECU configuration phase by adding the necessary service components and connecting them to the flattened set of atomic software components which require access to the services. As a consequence, compositions are not allowed to have ports for use with services. For more details about services, see AUTOSAR Services.

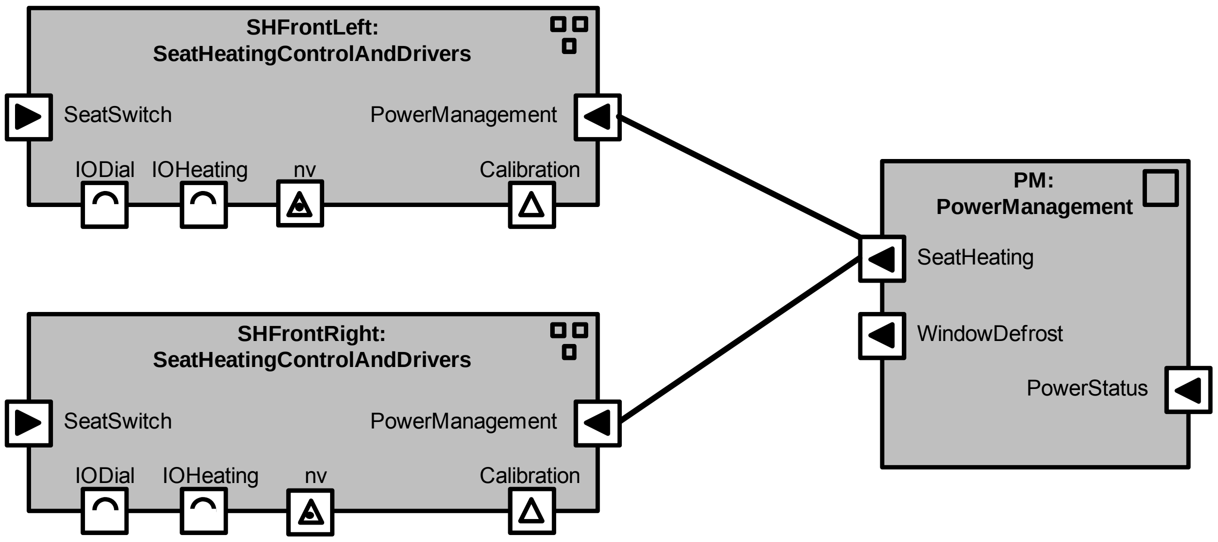


Figure 3.10: Example of the use of the Composition "SeatHeatingControlAndDrivers"

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.

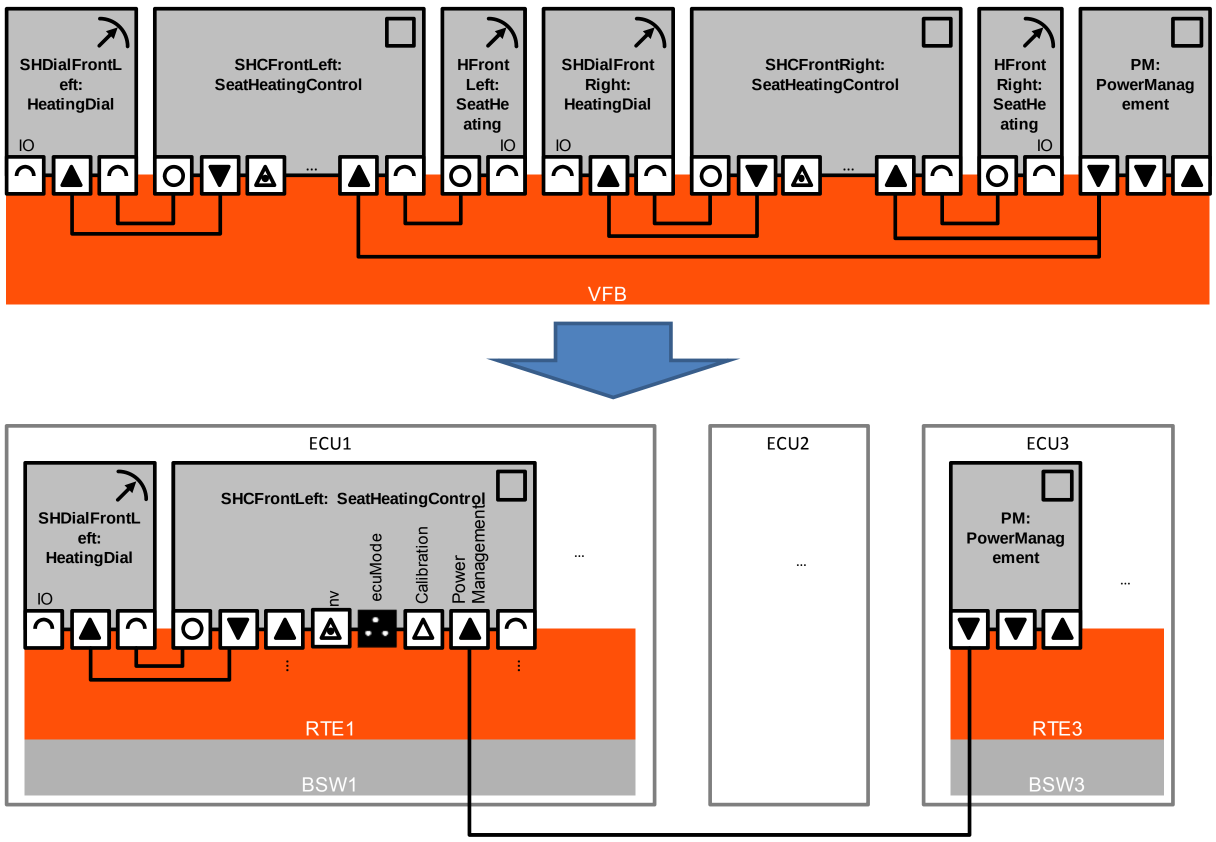

SHDialFrontLeft: HeatingDial
IO
SHCFrontLeft: SeatHeatingControl
HFrontLeft: SeatHeating
IO
SHDialFrontRight: HeatingDial
IO
SHCFrontRight: SeatHeatingControl
HFrontRight: SeatHeating
IO
PM: PowerManagement
VFB
ECU1
ECU2
ECU3
SHDialFrontLeft: HeatingDial
IO
SHCFrontLeft: SeatHeatingControl
nv
ecuMode
Calibration
Power Management
PM: PowerManagement
RTE1
BSW1
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 specification “Layered Software Architecture” [5].

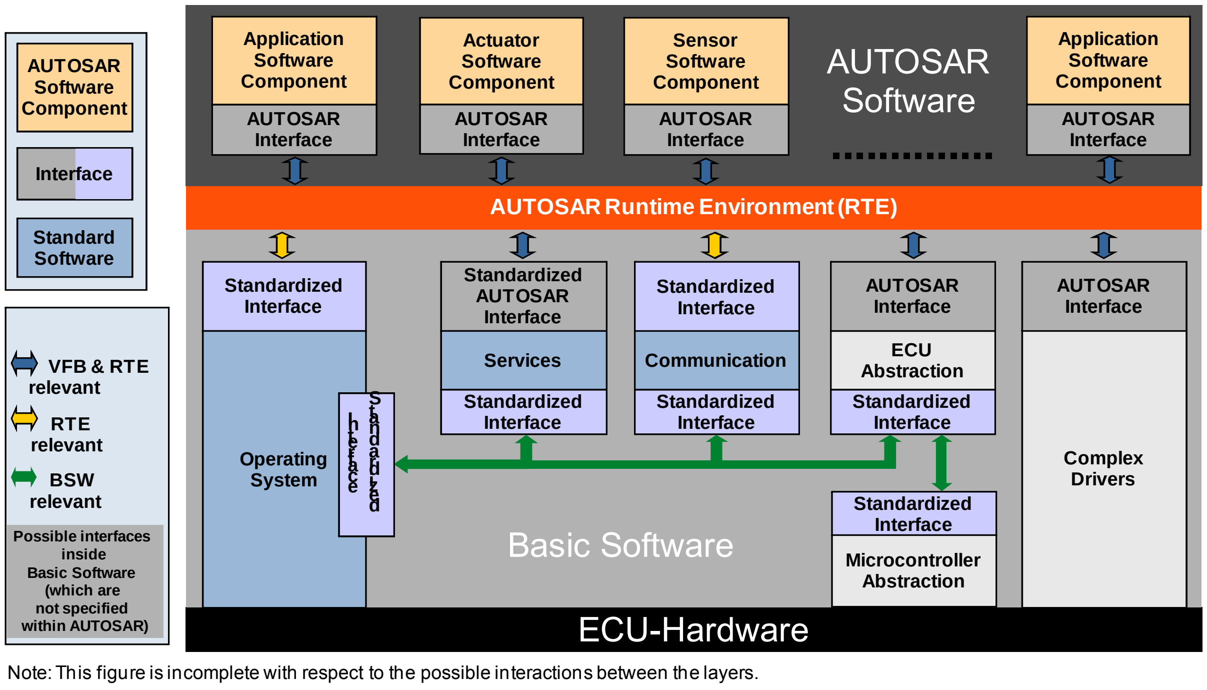


Figure 3.12: Component-View on the AUTOSAR layered software architecture

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”).

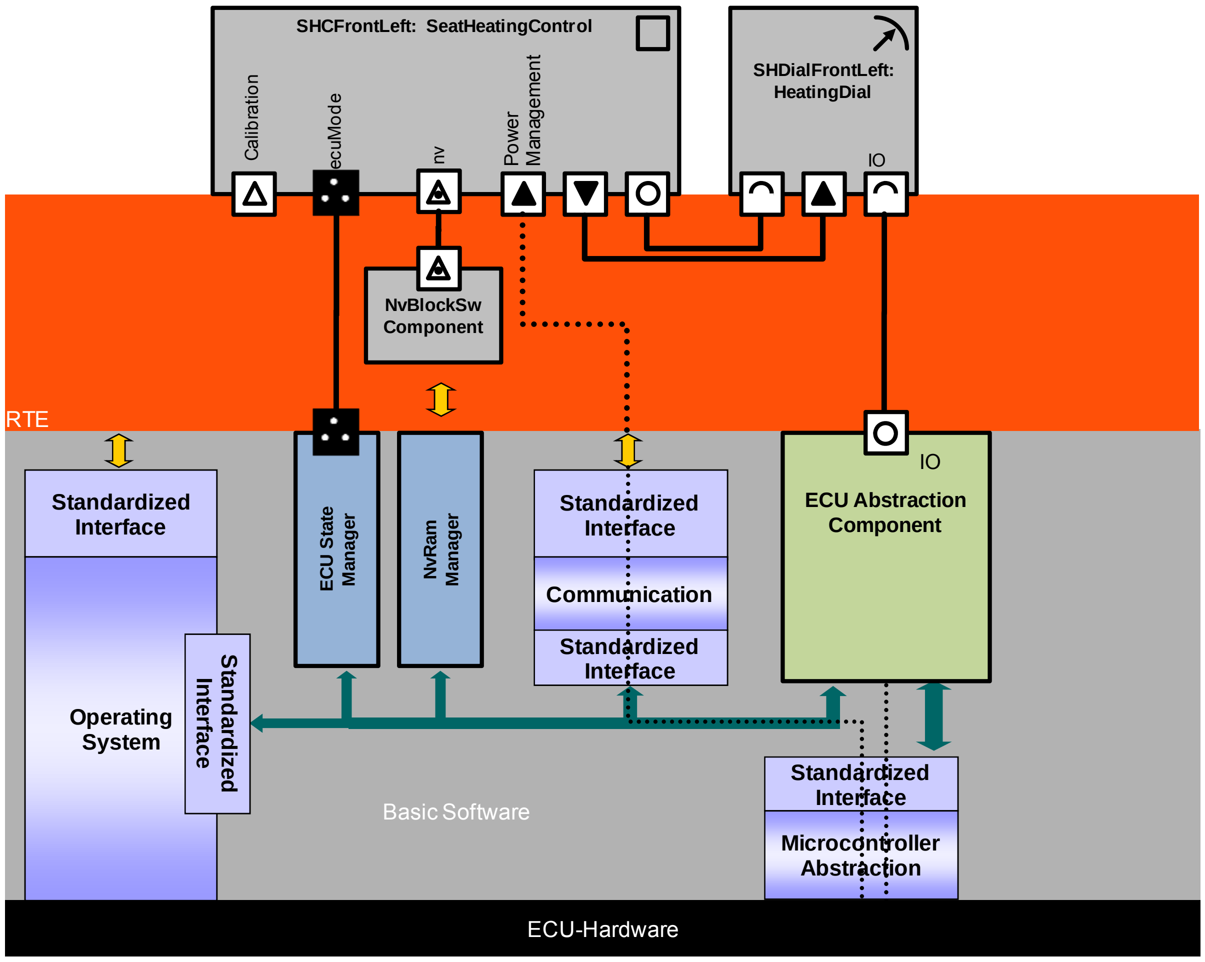
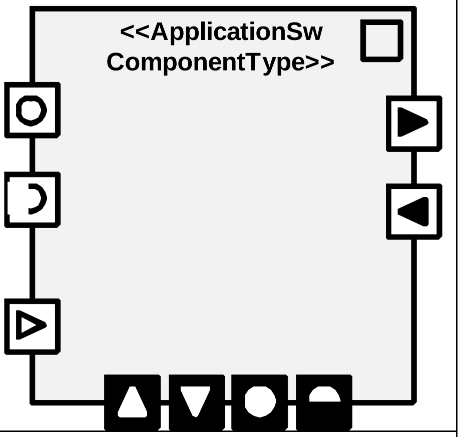
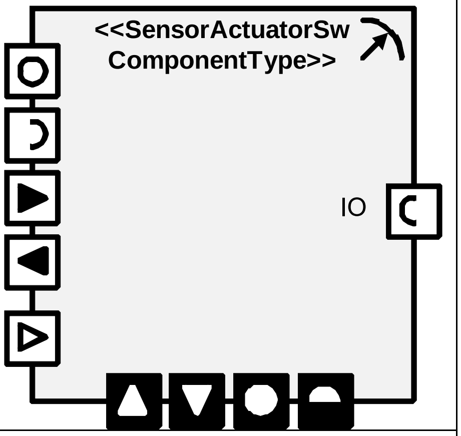
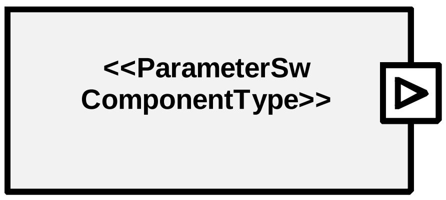
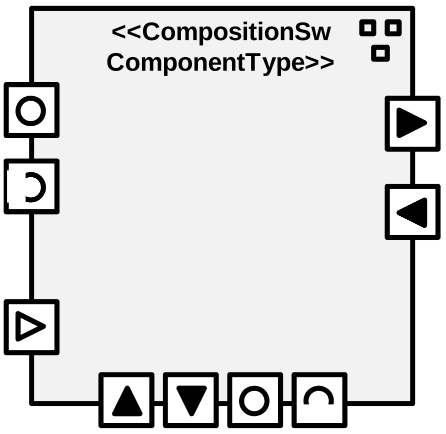


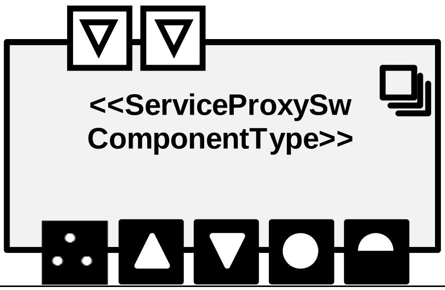
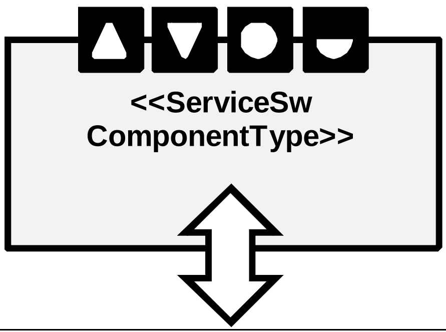
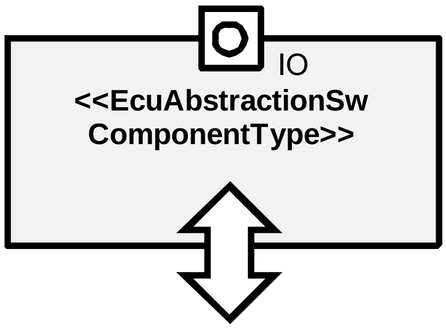
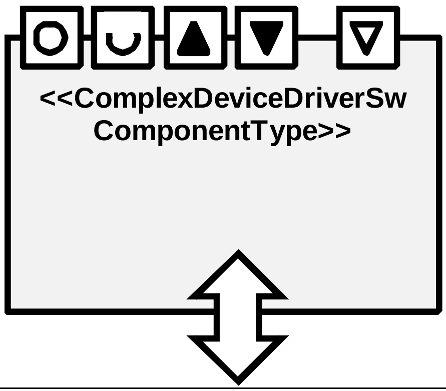
Figure 3.13: Example showing the relationship between the components mapped on an ECU and the ECU Software Architecture

3.7 Kinds of software 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. It 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.	
Parameter software component	A Parameter Software Component provides parameter values. These can be fixed data, const or variable. This Software Component allows for data access to either fixed data or calibration data. See chapter 10. .	
Composition software component	A Composition Software Component encapsulates a collaboration of Software Components, thereby hiding detail and allowing the creation of higher abstraction levels. Through delegation connectors a composition software component explicitly specifies, which ports of the internal components are visible from the outside. Composition Software Components are a specialized type of Software Components, e.g. they can be part of further Composition Software Components.	

Service Proxy software component	The Service Proxy SW Component is responsible for distribution of modes throughout the system. Once deployed each ECU should have a copy of every instance of this software component type. However at the VFB level only one is necessary.	
Service software component	A Service Software 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 software component	The ECU-Abstraction Software Component 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 software component	The Complex Device Driver Software Component 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 Software Component is described further in Section 6.5 Complex Device Driver.	

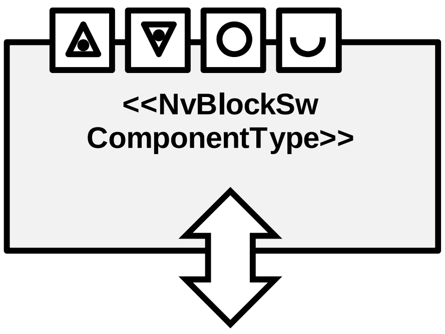
NVBlock software component	The NV Block Software Component allows SWC-S access to non volatile data. Specifically this block allows for the modeling of the NV data at the VFB level. It is the responsibility of the NV Block to map individual NV data elements to NV Blocks and to interact with the NV Manager in the BSW. The behavior of this component is to be generated based on the port services in the RTE.	

Table 3.4: Kinds of software 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)
    - Events related to physical occurrences (i.e. a triggered event).
- 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 expresses which information it requires from and provides to other components.

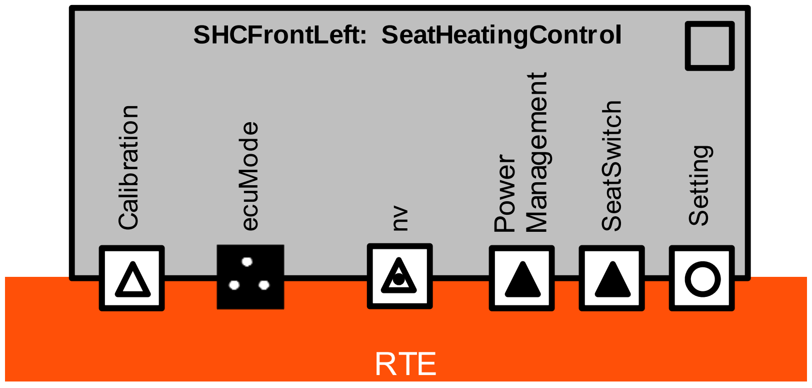


Figure 3.14: Component-view on the interaction between an atomic software component and the RTE on an ECU

However, the actual implementation of a component consists of a set of "runnable entities"[8] (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[9].

[8] 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".
[9] 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

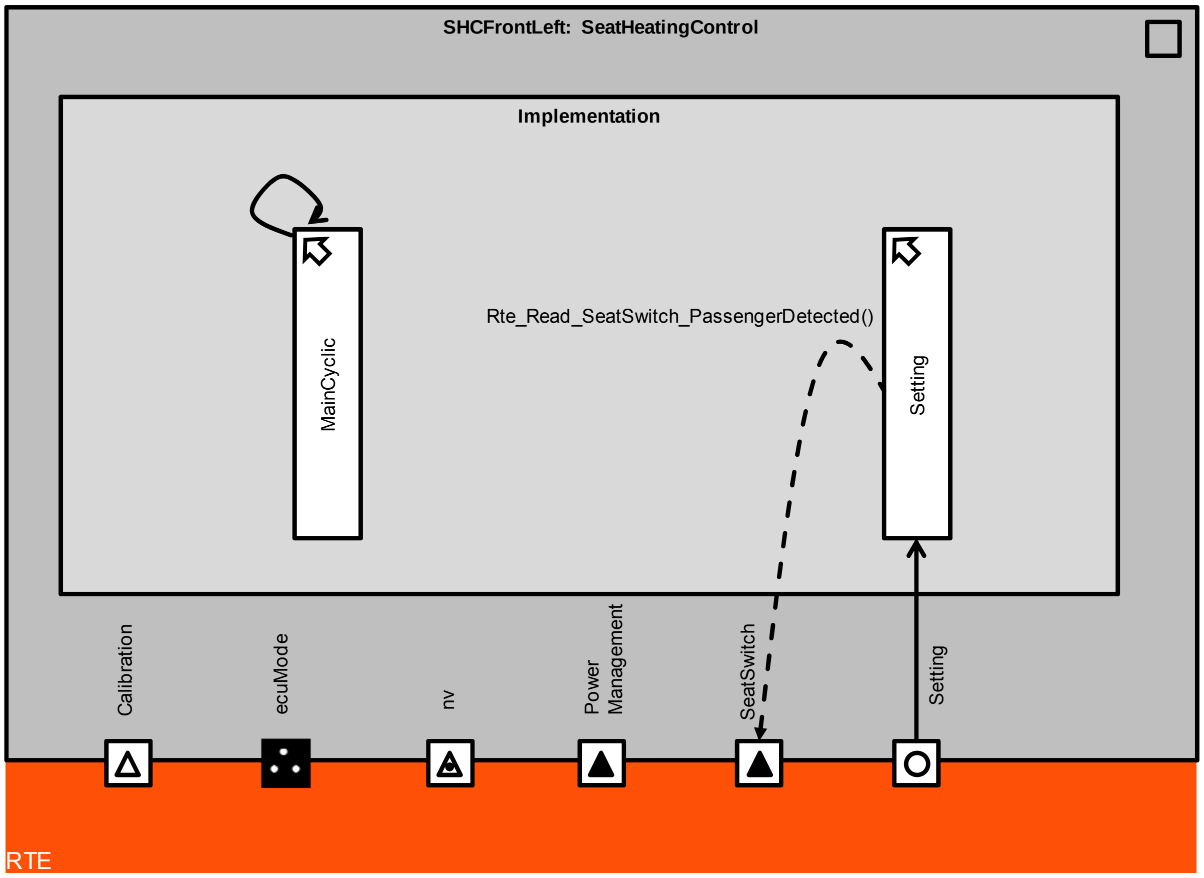


Figure 3.15: Implementation-view on the interaction between an atomic software component and the RTE on an ECU

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". The implementation of the runnables will use the operations provided by the RTE to actually for communication via the ports of the component. E.g. to access the information "PassengerDetected" provided to the component through the RPort "SeatSwitch" the runnable "Setting" will invoke the operation "Rte_Read_SeatSwitch_PassengerDetected()".
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 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.

A “runnable entity” runs in the context of a "task"[10]. 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 [6] 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 invoke operations
- Provide all other resources the component needs

3.9 Interface Conversion Blocks

When software components are developed by different organizations (e.g. two distinct suppliers delivering code to an OEM who integrates the SWCs) it may happen that two or more SWCs have the same engineering semantics but are represented with different data types. Instead of requiring the integrator to develop specific SWC conversion software the VFB will add a conversion block to a connector connecting Sender Receiver ports with mismatched interface definitions at the VFB level. The addition of this conversion block allows the designer to add which elements of the provided port map to the elements of the required port as well as provide the conversion semantics. In the RTE these mappings will be described with the PortInterfaceMappings. This construct maps an interface pair to the connection.

[10] 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.

[VFB140]「 If a P-port specified by a Sender Receiver Interface is connected to an R-port with an incompatible interface then a conversion block must be added for the connector to allow the designer to describe the conversion. Incomplete conversion will not be allowed」()

3.9.1 Supported Conversions and Mappings

3.9.1.1 Interface Element Mapping

In case two interfaces only differentiate in the shortnames of their elements, then a mapping can be provided which maps the elements of the one interface to the elements of the other interface.

3.9.1.2 Linear Data Conversion

If the elements of two interfaces are logically equivalent but the range and resolution are different, then the linear conversion factor can be calculated out of the semantical information of the elements. In this case the data semantics is described using a CompuMethod with category IDENTICAL, LINEAR, SCALE_LINEAR or SCALE_LINEAR_AND_TEXTTABLE, where the

- IDENTICAL category means that the value of the physical representation is equal to the internal representation and the
- LINEAR, SCALE_LINEAR or SCALE_LINEAR_AND_TEXTTABLE categories mean that the internal representation is calculated out of the physical representation by means of a linear formula (factor * external value + offset) per range in one or more ranges (SCALE_LINEAR only).

[VFB141] 「A conversion block involving either IDENTICAL, LINEAR, SCALE_LINEAR or SCALE_LINEAR_AND_TEXTTABLE data types shall use the COMPU-METHODS for the respective data types to determine the conversion function.」()

The following examples show the conversion of data that is described using CompuMethods with category LINEAR and IDENTICAL:

1) A software component (A) that provides the vehicle speed in m/s with resolution 0,1 m/s can be connected with a component (B) that requires the vehicle speed in m/s with a resolution of 0,01 m/s if both components assume a **linear** relation between physical representation and internal representation. The foll
internal (A) = 10 * physical as m/s
internal (B) = 100 * physical as m/s

internal (B) = 100 * physical as m/s
= 100 * (internal (A) / 10)
= 10 * internal (A)

Example: Component A provides the value 100 (internal representation for 10 m/s). Multiplying the value with 10 we get the value 1000 as input for component B (internal representation of 1000 in component B corresponds to 10 m/s)

2) A special case of data scaling is the conversion of units: Software component (A) that provides the vehicle speed in m/s can be connected with a component (B) that requires the vehicle speed in km/h if both components assume a **linear** or **identical** relation between physical representation and internal representation.

internal (A) = physical as m/s
internal (B) = physical as km/h

internal (B) = physical as km/h
= 3,6 * physical as m/s
= 3,6 * internal (A)

Example: Component A provides the value 10 (internal representation for 10 m/s). Multiplying the value with 3,6 we get the value 36 as input for component B (internal representation of 36 corresponds to 36 km/h which is equivalent to 10 m/s)

3.9.1.3 Data Mapping

In case the data semantics is described using a list of values (CompuMethod with category TEXTTABLE) or partially described using a list of values (CompuMethod with category SCALE_LINEAR_AND_TEXTTABLE), then an explicit mapping needs to be given for each individual value.

[VFB142] 「A conversion block involving TEXTTABLE or SCALE_LINEAR_AND_TEXTTABLE data types shall use explicit mapping of each RPort table element to a PPort table element.」()

3.9.1.4 Mixed Conversion

It is possible in a conversion block to mix both linear conversion and texttable mappings (SCALE_LINEAR_AND_TEXTTABLE).
An example would be a conversion block consisting of an input value of type uint8 which is linearly converted in the range 0..200 and has 2 texttable mappings for the values 254 “SensorNotAvailable” and 255 “SensorFault”.

3.10 Variant Handling

To support variation in automotive applications AUTOSAR has a mechanism referred to as variant handling. This allows designers at many levels to put together a super set of functionality and choose which actual pieces of this functionality will be enabled in a specific variant. The place in the design where a choice is given between 2 or more variants is called a variation point. The time at which a choice must be made is called the latest binding time. Binding earlier is always allowed.

3.10.1 Binding Times

AUTOSAR supports several discreet binding times:

- System Design
- Code Generation
- Pre Compile
- Link Time
- Post Build

Although variability could exist at function design time and run-time AUTOSAR explicitly prohibits the later and does not provide support for the function design time.

3.10.2 Choosing a Variant

To choose a variant the AUTOSAR designer must assign no later than the required binding time one of a predefined set of values to a Software System Constant or to a Post Build Variant Criterion. The Post Build Variant Criterion is used for enabling Post Build binding times while the Software System Constant can be used for everything that has a latest binding time of Link Time.

By assigning a value to either a Software System Constant or Post Build Variant Criterion the AUTOSAR system can determine which variant is enabled for each Variation Point in the design by evaluating either a Software System Dependant Formula (uses System Constants to determine if a Variation Point is enabled or disabled) or by evaluating one or more a Post Build Variant Conditions (uses Post Build Variant Criterions to determine if a Variation Point is enabled or disabled). If the Variation Points Formula or Condition evaluates to true then the element in the design which was conditional upon the Variation Point will exist in the design.

Typically designers will define collections of validated assignments for Software System Constants and Post Build Variant Criterions. These collections of value assignments are also known as Predefined Variants. Predefined Variant Sets are typically defined at a composition level like a subsystem or system design. A complete variant for a system therefore typically exists of a collection of Predefined Variants binding every Variation Point in the system.

3.10.3 Variability

Although variability exists within the internal behavior of Software Components from a VFB perspective only three elements of variability are of interest:

- Software Component Variability
- Port Variability

- Connector Variability.

3.10.3.1 Software Component Variability

The existence of a Software Component either Atomic or Composition can be subjected to the existence of a Variation Point. If a Variation Point exists and its conditions (see section choosing a Variant) evaluate to true then the Software Component exists and its behavior will be scheduled and its ports produce output. If the Component however is removed from a composition (I.e. application or system design) then all Software components which are connected to the removed Software Component will have ports which will be considered unconnected and will behave as unconnected ports (see section Unconnected Ports for more details) and non of the behavior of the removed component will execute. Software Components variability in a Composition can be bound as late as Post Build.

3.10.3.2 Port Variability

Ports on a Software Component can also be subjected to variability. However their latest binding time is Pre Compile time and as such their variability can only be constrained using Software System Constants. If a Port is removed from the design then any connecting port must behave as an unconnected port. In a properly configured system if a Port is “disabled” the connector connecting to this port should also be subjected to the same variability conditions.

3.10.3.3 Connector Variability

A connection between two ports can be subjected to variability with a binding time of Post Build. If a connector is “disabled” then the two ports at either end of the connector must behave as unconnected ports.

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.

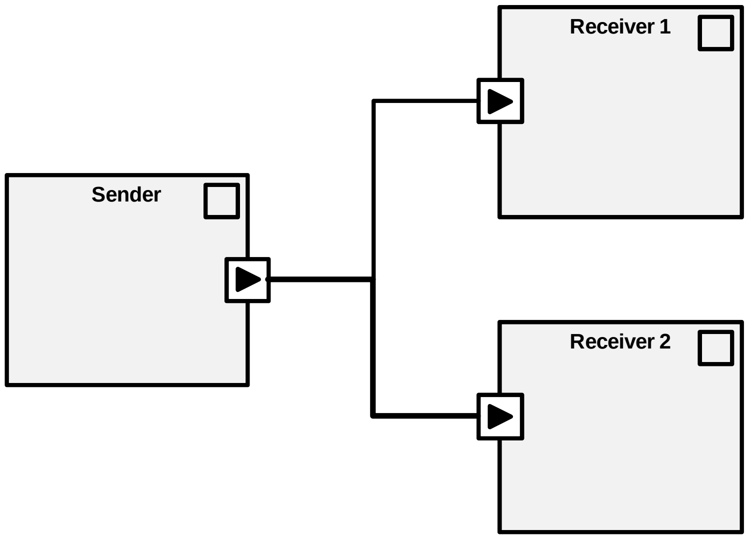


Figure 4.1: Example of sender-receiver communication at VFB level

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 [6]	Description	Kind of data-element or modeGroup		
			data	event	mode
INIT_VALUE	attribute “initValue” of “UnqueuedSenderComSpec”	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[11]	not available[12]

[11] The initial condition of a queued data-element is the empty queue

CAN_INVALIDATE	attribute "canInvalidate" of "NonqueuedSenderComSpec"	In case this feature is used, the sender can invalidate a data-element.	optional	not available	not available
MODE_QUEUE_LENGTH	"queueLength" of ModeSwitchSenderComSpec	This attribute defines the size of the input queue of the of mode switch notifications to a mode machine.	not available	not available	required
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 "VariableDataPrototype"	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 the "Software Component Template" [6] and the "SWS RTE" [7].

[12] The initial mode is defined as part of the ModeDeclarationGroup

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 "NonqueuedReceiverComSpec"	A receiver can optionally specify its own initial value, which overrides the initial value of the sender.	optional	not available[13]	not available[14]
RECEIVE_INVALID	"handleInvalid" in "NonqueuedReceiverComSpec"	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 "NonqueuedReceiverComSpec"	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

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

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
RECEIVE_EVENT	“DataReceivedEvent” and “SwcModeSwitchEvent”	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 “VariableDataPrototype”	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
SW_IMPLEMENTATION_POLICY	"swImplPolicy"	When using a parameter interface one can type the mechanism for access of the parameters. This will allow for precompile time and compile time optimization when dealing with fixed data exchange	optional	not available	not available

Table 4.2: Communication Attributes for a Receiver

Details can be found in the "Software Component Template" [6] and the "SWS RTE" [7].

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.

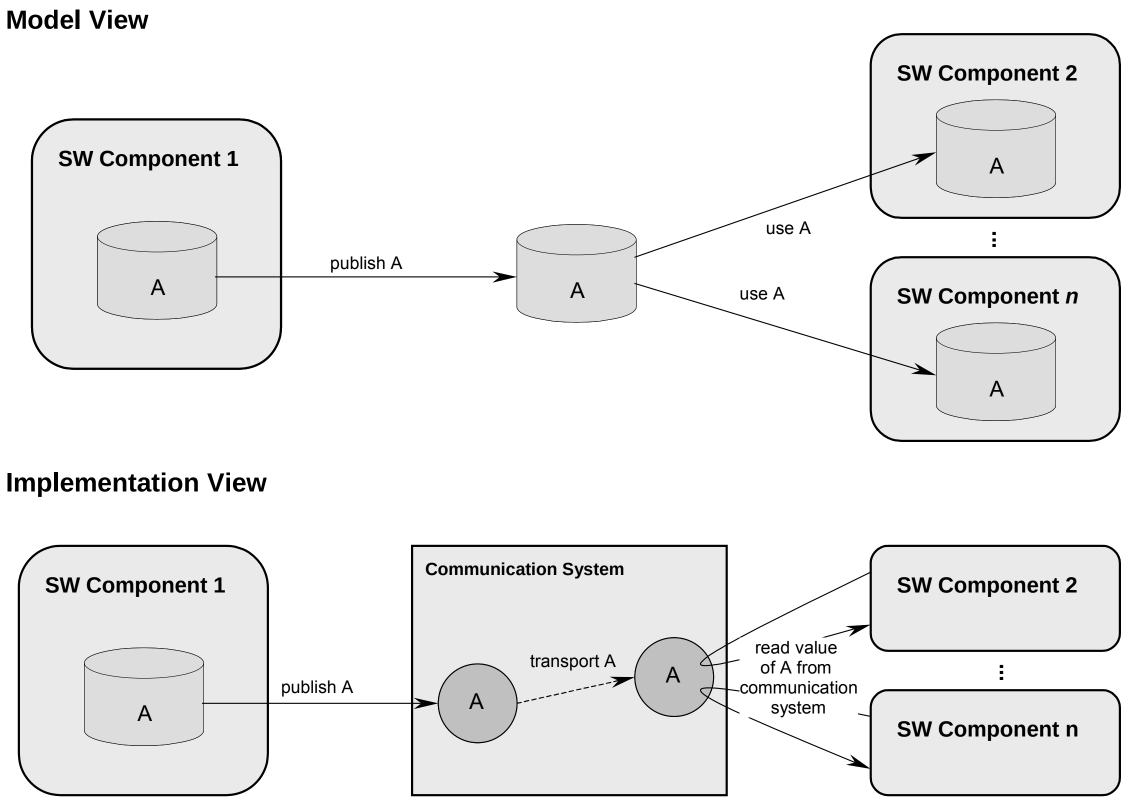


Figure 4.2: ”last-is-best” semantics. The upper part of this figure shows the model view of ”last-is-best” semantics. The lower part shows the implementation view of this pattern.

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.

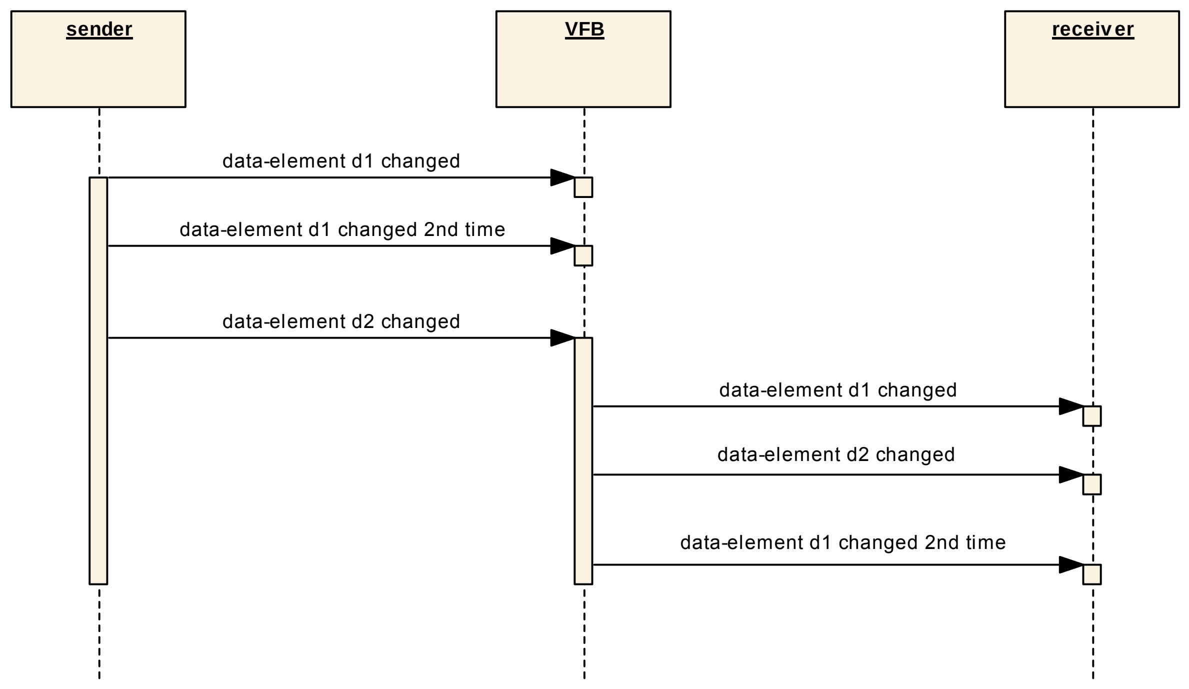


Figure 4. 3: concurrency and ordering within a sender-receiver connector

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[15] 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).

[15] 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.

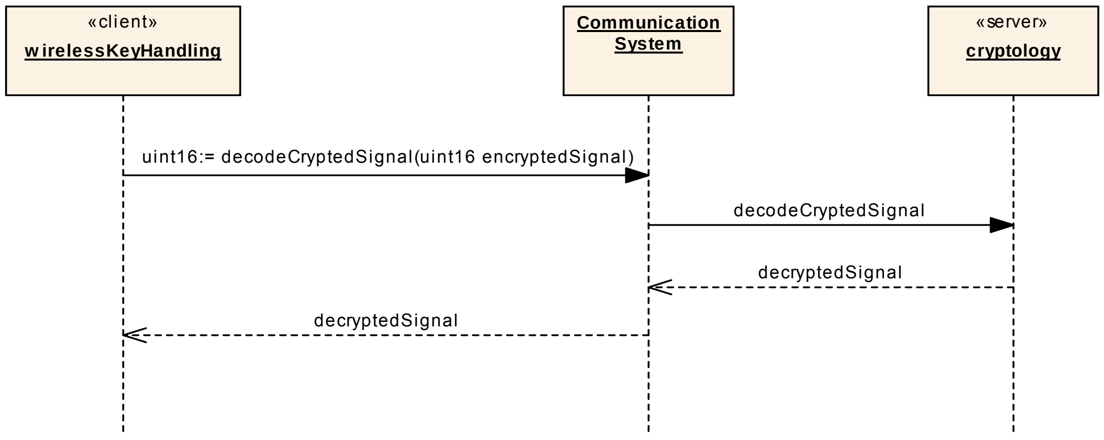


Figure 4.4: Example of a synchronous client-server communication: decoding of encrypted wireless-key data (immobilizer).

AUTOSAR defines a very simple, static n:1 client-server mechanism (n clients and 1 server, with n ≥ 0)[16]. 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.

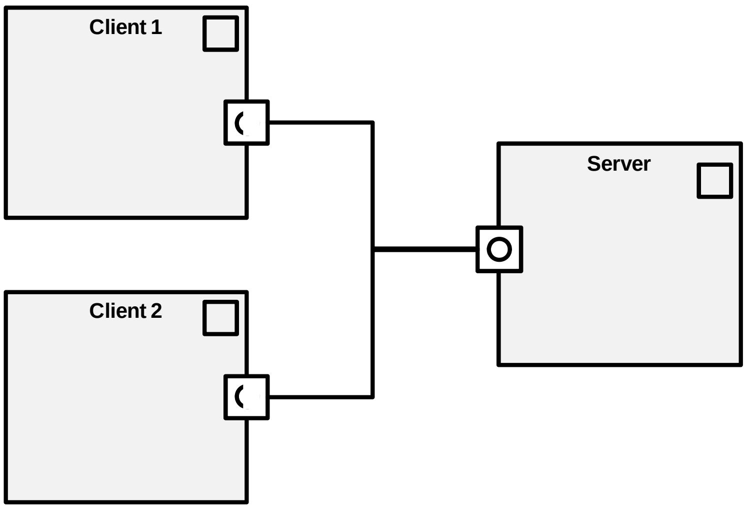


Figure 4.5: Client-server communication in the VFB View

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

[16] 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.[17]

[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.

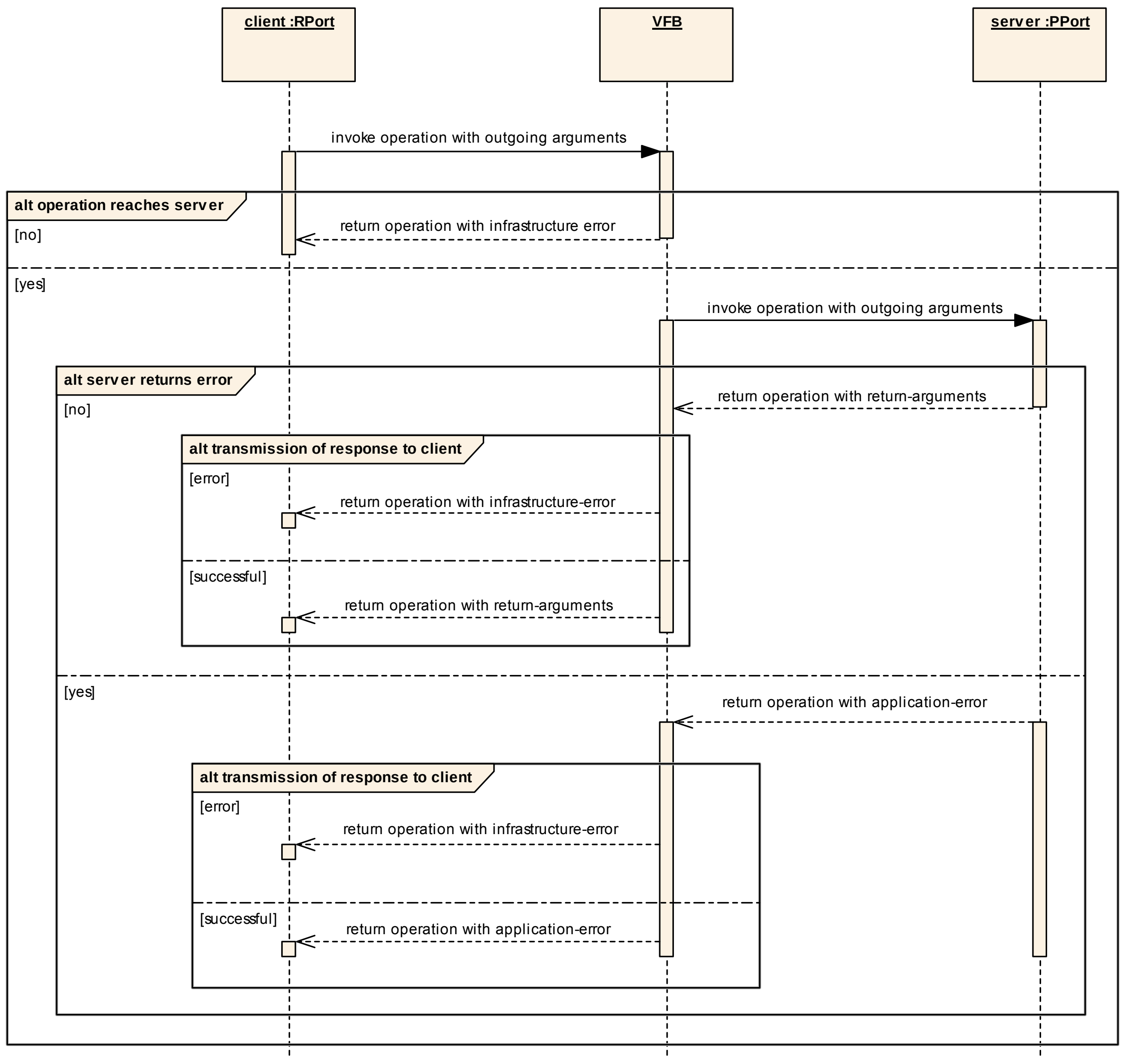

client :RPort
VFB
server :PPort
invoke operation with outgoing arguments
alt operation reaches server
[no]
return operation with infrastructure error
[yes]
invoke operation with outgoing arguments
alt server returns error
[no]
return operation with return-arguments
alt transmission of response to client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

[17] Details about the data-types supported by AUTOSAR in arguments can be found in [SW-C 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 [6]	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. 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. 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 [6]	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.

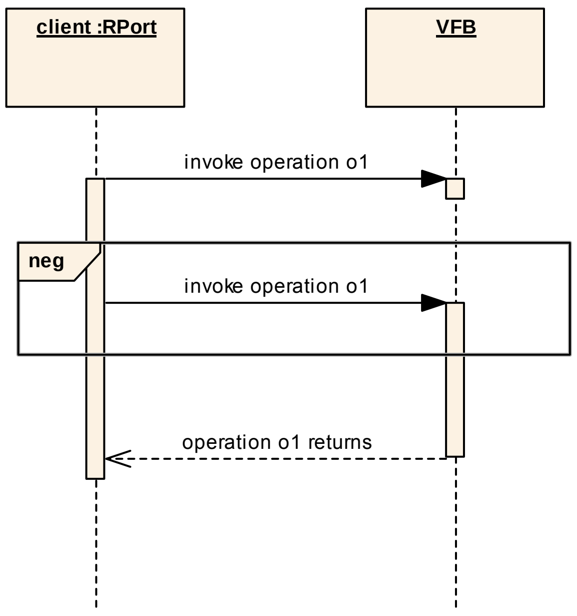


Figure 4.7: Concurrent invocation of the same operation is not allowed

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⌋ ()

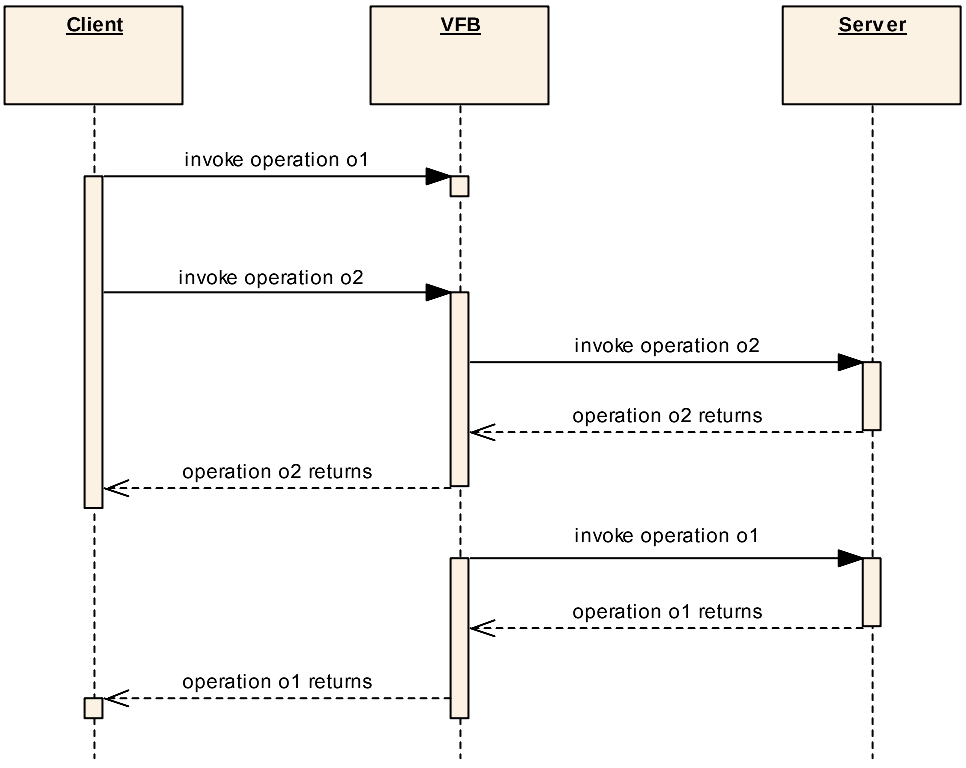


Figure 4.8: The VFB does not support ordering between different operations

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[18].
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[19].

[18] For future extensions like “dynamic components” and “dynamic communication” communication partners have to provide means to be identified on application level.
[19] 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 Extensions

The research field of real time systems offers a variety of timing models and specification techniques. This section just serves as a high level introduction to the “AUTOSAR Specification of Timing Extensions” [8] and only has the intent to make the reader aware of a different and more detailed document which addresses the concerns of modeling time.

5.1 Main Purpose of Timing Extensions for AUTOSAR

Compared to the specification of a system's functional behavior, the specification of its timing behavior requires additional information to be captured. Not only the eventual occurrence of events but also their exact timing or the concurrency of various events become important. Therefore, in the specification of timing extensions for AUTOSAR, the event is the basic entity. It is used to refer to an observable behavior within a system (e.g. the activation of a RunnableEntity, the transmission of a frame etc.) at a certain point in time.

Having to deal with different abstraction levels and views, and in order to avoid semantic confusion with existing concepts, a new abstract type TimingDescriptionEvent is introduced as a formal basis for the timing extensions. Depending on the concrete model entity and the associated observable behavior, specific timing events are defined and linked to the different views.

For the analysis of a system's timing behavior usually not only single events but also the correlation of different events is of interest. To relate timing events to each other, a further concept called TimingDescriptionEventChain is introduced. Hereby, it is important to note that for the events referred to within an event chain a functional dependency is implicitly assumed. This means that an event of a chain somehow causes subsequent chain events.

Based on events and event chains, it is possible to express various specific timing constraints derived from the abstract type TimingConstraint. These timing constraints specify the expected timing behavior. As timing constraints shall be valid independently from implementation details, they are also expressed on a abstract level by referencing the above introduced formal basis of TimingDescriptionEvents and TimingDescriptionEventChains.

Thus, by means of events, event chains and timing constraints defined on top of these, a separate central timing specification can be provided, decoupling the expected timing behavior from the actually implemented behavior. This approach supports timing contracts for AUTOSAR systems in a top-down as well as bottom-up approach.

5.2 Timing in different phases of the AUTOSAR methodology

Several timing views can be applied in the different phases of the AUTOSAR methodology which provides several well defined process steps, and furthermore artifacts that are provided or needed by these steps. Five different timing views can be identified:

- **VfbTiming** – this view deals with timing information related to the interaction of SwComponentTypes at VFB level.
- **SwcTiming** – this view deals with timing information related to the SwcInternalBehavior of AtomicSwComponentTypes.
- **SystemTiming** – this view deals with timing information related to a System, utilizing information about topology, software deployment, and signal mapping.
- **BswModuleTiming** – this view deals with timing information related to the BswInternalBehavior of a single BswModuleDescription.
- **EcuTiming** - this view deals with timing information related to the EcucValueCollection, particularly with the EcucModuleConfigurationValues.

For each of these views a special focus of timing specification can be applied, depending on the availability of necessary information, the role a certain artifact is playing and the development phase, which is associated with the view.

The “AUTOSAR Specification of Timing Extensions” [8] provides a concept for the description of timing relevant information in AUTOSAR.

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):

- o if the µC changes, it is possible to reuse the ECU Abstraction, the sensor-actuator software-component and the application software-component
- o if the ECU changes, it is possible to reuse the sensor-actuator software-component and the application software-component
- o 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)

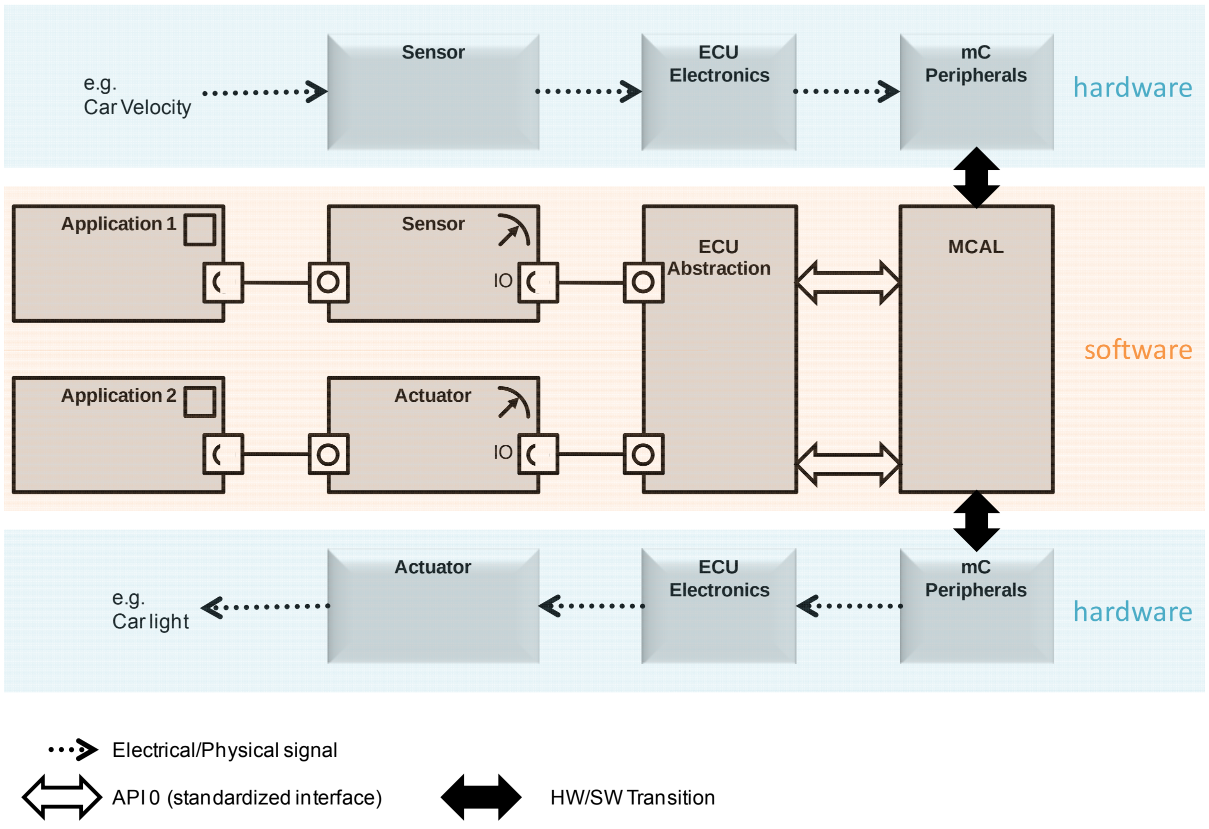


Figure 6.1: Signal conversions between physical signals and software signals

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

The MCAL is available on each standard microcontroller.

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

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.

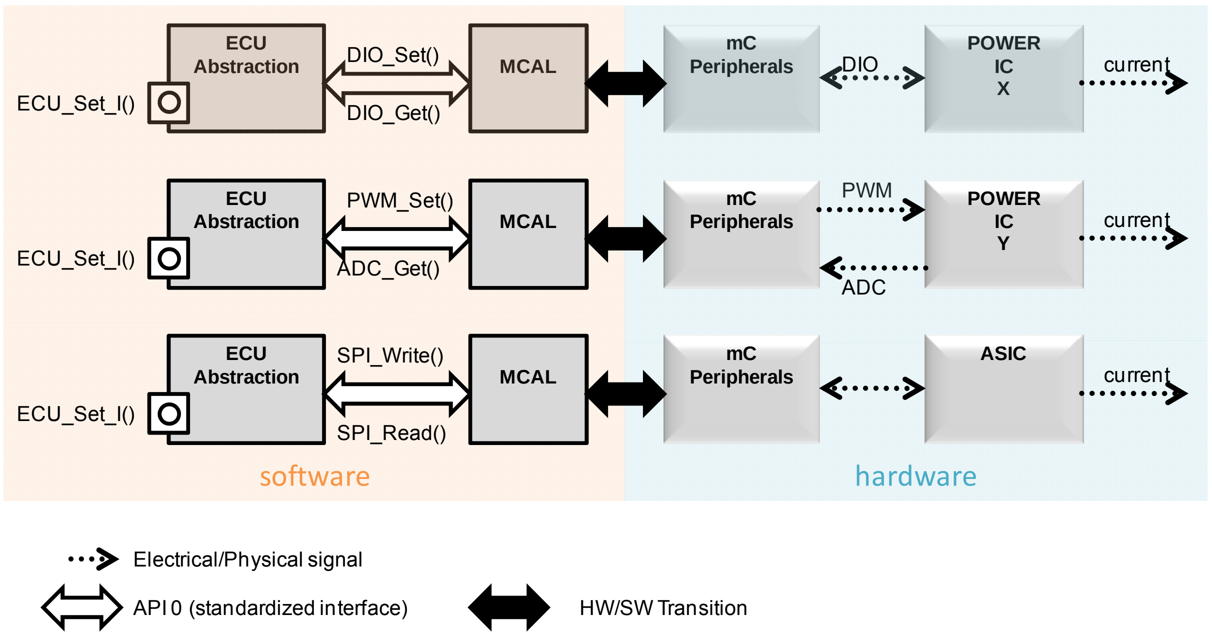


Figure 6.2: example "ECU_Set_I" for the ECU abstraction

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.

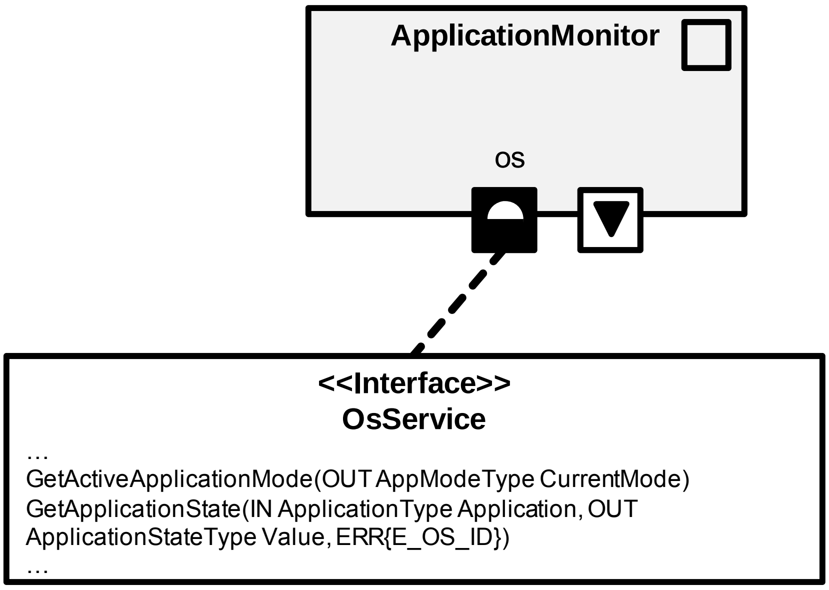

ApplicationMonitor
os
<<Interface>>
OsService
…
GetActiveApplicationMode(OUT AppModeType CurrentMode)
GetApplicationState(IN ApplicationType Application, OUT ApplicationStateType Value, ERR{E_OS_ID})
…

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 ApplicationMonitor has a port typed with the interface OsService. Since this client-server interface contains operations like GetActiveApplicationMode or GetApplicationState, the software component ApplicationMonitor is able to query the Os about the OsApplication states or the Os start mode.
Figure 8.4 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 [7])
- 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 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 8.4 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 Distribution of Requests to Remote Services

A direct communication from an application software component to a remote ECU's AUTOSAR service is not possible. On the other hand, the concept of application and vehicle mode management requires the distribution of mode requests from one mode requestor to the service of a Basic Software Mode Manager (BswM) on every ECU. To distribute the requests, service proxy SW components are used.

The service proxy SW component is similar to an application SW component. But, the same service proxy SW component instance is copied during the system design to several ECUs while an application SW component instance is mapped to exactly one ECU in the system.

As a consequence, a connection between an application software component and a service proxy SW component that is shown as 1:1 connection in the VFB will be a 1:n connection in the system. This allows the distribution of a request to several ECUs.

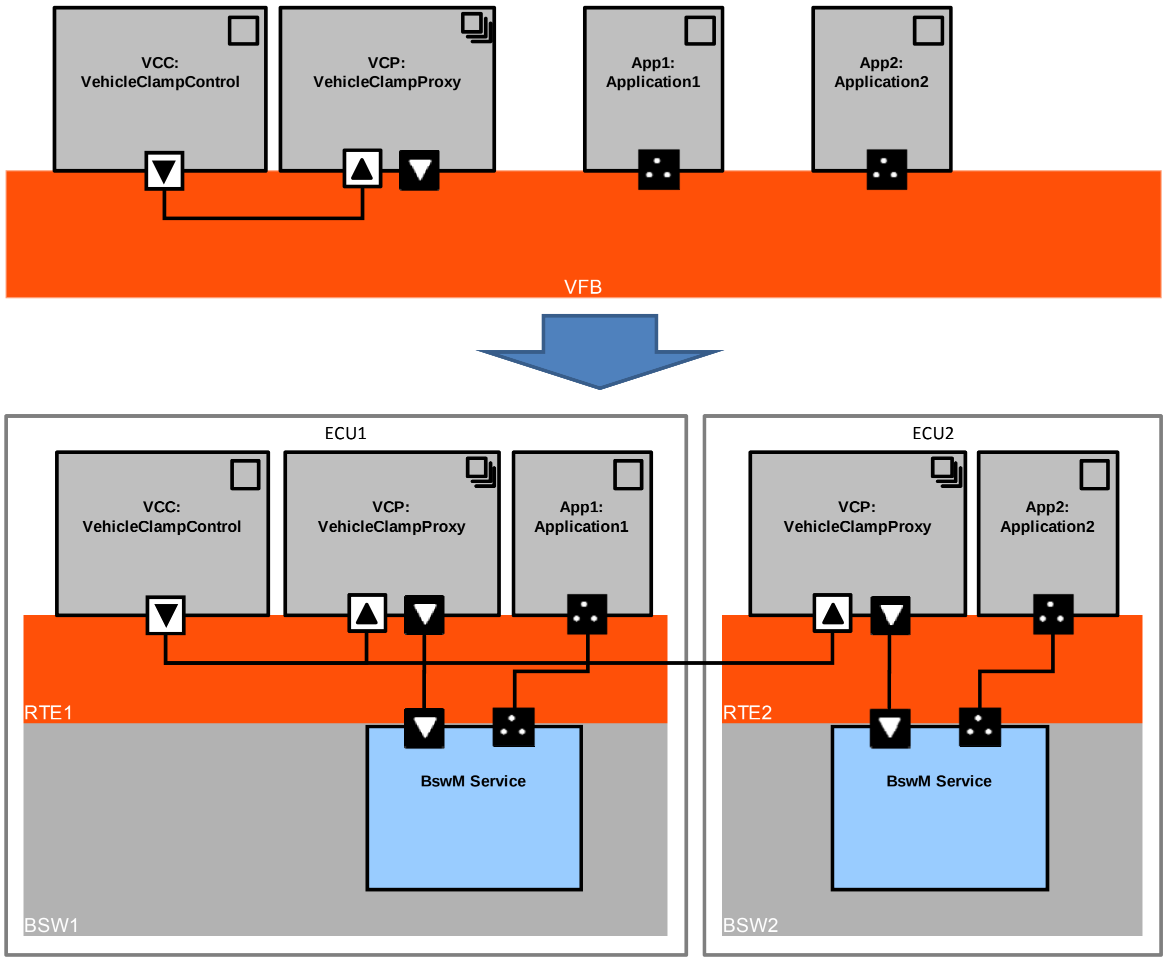


Figure 7.2: Example for distributing a mode request from a VehicleClampControl to the BswM of several ECUs

7.2.4 Platform dependent types

Many data types within the Basic software are platform dependent to gain efficiency. For example: the type of IDs can depend on the entities to be handled within a specific ECU, which would 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.

7.2.5 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 the "Software Component Template" specification [6].

7.3 List of Services

As of AUTOSAR Release 4.0 services of the following BSW modules are available:

1. NVRAM Manager – NvM
2. Communication Manager – ComM
3. Diagnostic Communication Manager – Dcm
4. Diagnostic Event Manager – Dem
5. Function Inhibition Manager – Fim
6. ECU State Manager – EcuM
7. Watchdog Manager – WdgM
8. Development Error Tracer – DET
9. Crypto Service Manager – Csm

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 mode switch notification mechanism is used to exchange modes between components. A mode switch interface includes a so called "ModeDeclarationGroup".
Figure 8.1 shows an example of the definition of the mode switch interface "ECUMCurrentMode" containing a single reference to the ModeDeclarationGroup "ECUMMode".

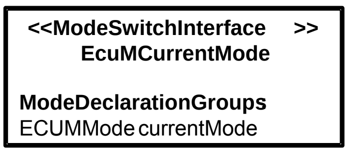


Figure 8.1: Example of a Sender-Receiver Interface "ECUMCurrentMode" with a single ModeDeclarationGroup

The ModeDeclarationGroup is a set of ModeDeclarations. Within the definition of the group, one ModeDeclaration describes the initial mode that is assumed at startup. 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 mode PPort of a component」()

[VFB116]「 At configuration time, the initial mode of each ModeDeclarationGroup in a mode switch interface is known」()

[VFB112]「 At configuration time, it is known which ModeDeclarationGroup a mode switch interface contains」()

[VFB114]「 At configuration time, the modes of each ModeDeclarationGroup in a mode switch interface are known」()

8.3 Communicating modes

Modes are transmitted via the mode switch notification mechanism.
There will be software-components that have PPorts typed by mode switch interfaces. 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.

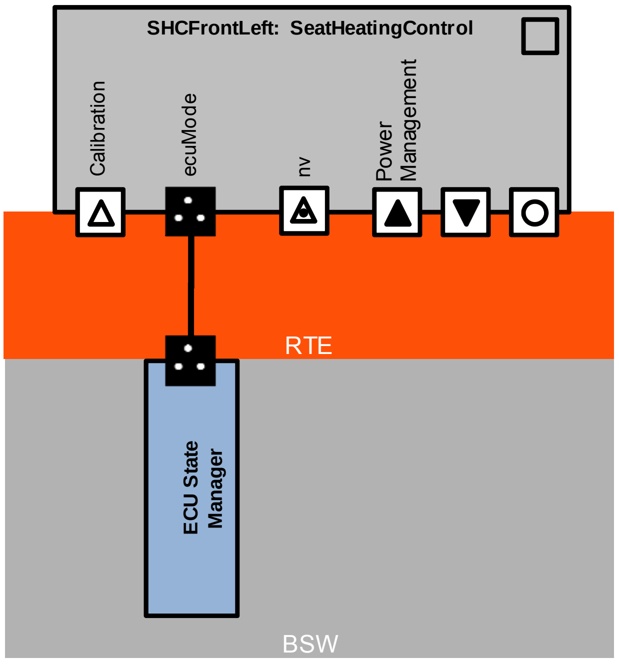


Figure 8.2: Example of a the communication of a mode from the "ECU State Manager" Service-component to an application software-component

For mode switch interfaces, only 1:n communication (1 mode manager and n mode users, with n ≥ 0) is possible. The single mode manager owns the current mode of the ModeDeclarationGroup. The users are informed of any mode switch of the manager.
For the mode managers of the AUTOSAR basic software, there is typically for each mode switch based service also a sender receiver based service to request a mode. E.g., for each ComM user one mode switch interface indicates the currently available communication mode and a sender receiver interface is used to request the desired communication mode. In this pattern there is usually one mode requestor that is at the same time a mode user. Figure 8.3 shows this pattern for the ComM.

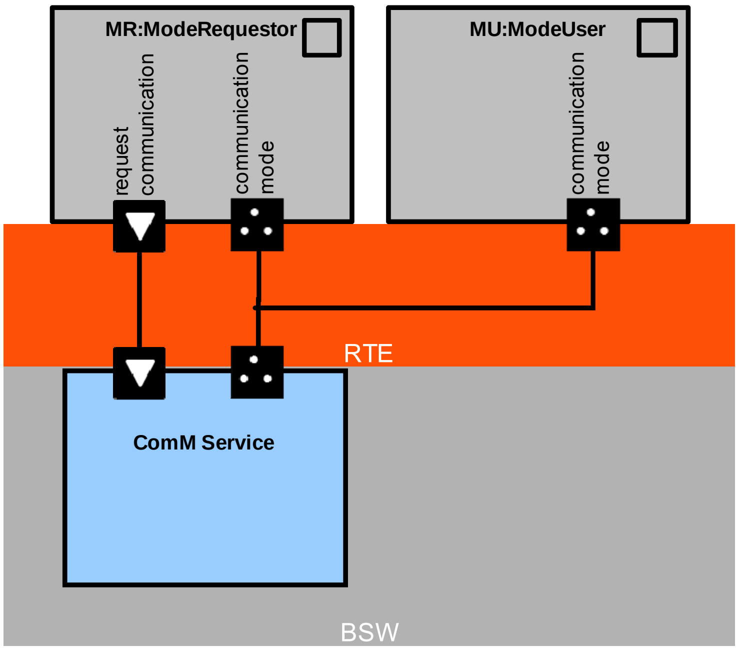


Figure 8.3: Example of a the communication of a mode from the "ECU State Manager" Service-component to an application software-component

Due to the strong synchronization between a mode manager and the mode users, mode switch communication is only supported in ECU local communication. For a mode management that spans several ECUs, a communication pattern including service software proxy components for the distribution of mode requests and the BswM for the switching of modes on each ECU is recommended (see section 7.2.3).

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 mode switch interface which references the appropriate ModeDeclarationGroup. The state of the mode managers will be sent to other component using sender-receiver communication.
Optionally, a mode manager can have an RPort typed by a sender receiver interface with a data element that is mapped to the same ModeDeclarationGroup to receive mode requests from a mode requestor.

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 mode switch interface which references the appropriate ModeDeclarationGroup.
Figure 8.4 shows an example whereby the mode switch 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). The component could disable the execution of certain runnables during the mode STARTUP_SHUTDOWN and start initialization runnables on the transition to the mode RUN.

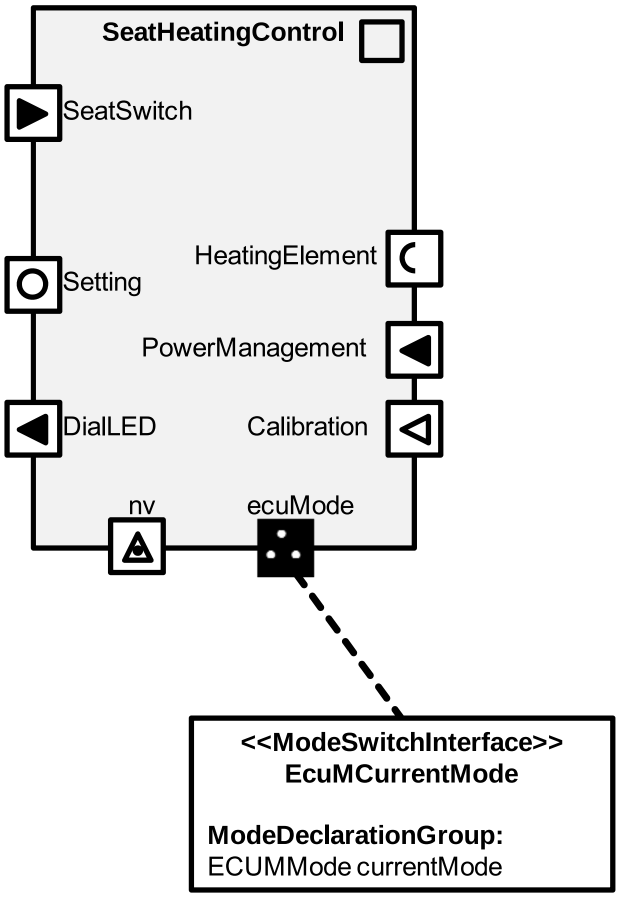


Figure 8.4: Example showing the use of the mode switch Interface "ECUMCurrentMode" to type the Port "ecuMode" of the component "SeatHeatingControl"

[VFB117]「At configuration time, it must be known which mode switches, the receiver of a ModeDeclarationGroup in a mode switch interface wants to be informed of」()

[VFB119]「The transition of modes received from the same ModeDeclarationGroup instance of a mode manager shall be perceived synchronously by all mode users」()

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 Port Groups

There is a natural hierarchical grouping of ports given by the aggregation of port prototypes in software components. In addition, AUTOSAR supports alternative grouping of ports according to other aspects of the vehicle system software. This is expressed by port groups. The main use case for port groups is to express the required communication resources during a certain mode of operation like a limp home mode or a diagnostic mode. These modes are usually orthogonal to the decomposition in components and sub-components.

A port group has the following features:

- aggregated to a software component type
- list of require and provide port prototypes of the software component
- reference to the sub component port groups that are merged into the port group.

As a practical use case, a port group can reflect a ComM user in the VFB. The configuration of communication channels associated with a ComM user can be extracted from the VFB model automatically.

There can be independent mode managers for terminal clamp control, for power saving, for diagnostic mode, etc. Each of these mode mangers can also have independent partially overlapping port groups.

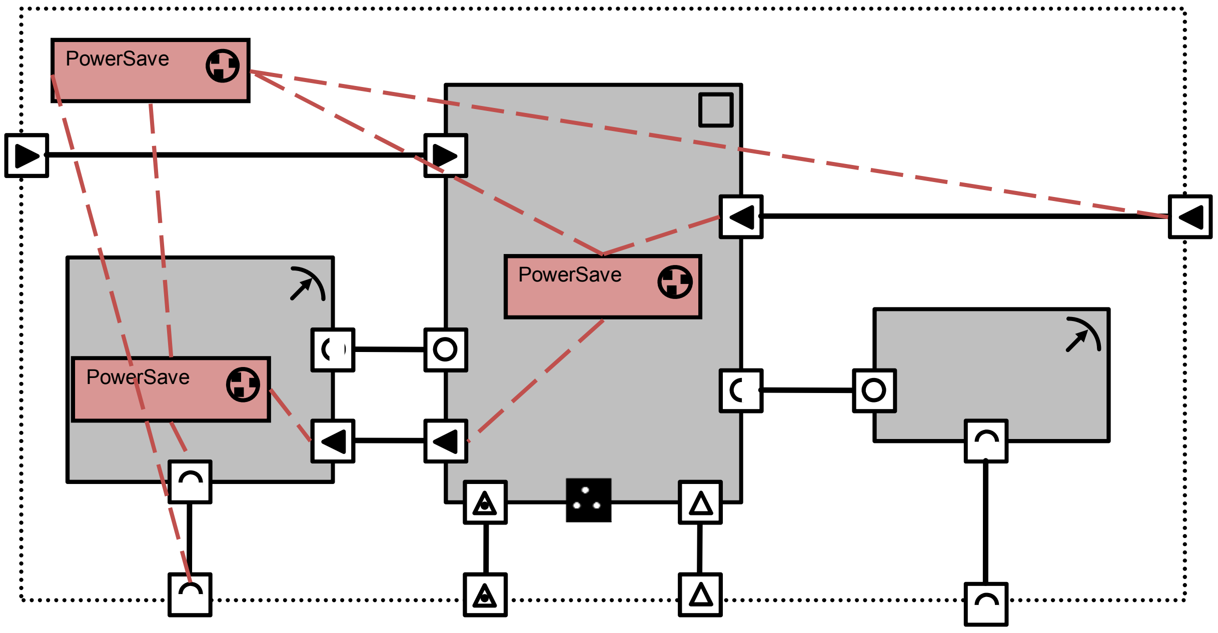


Figure 9.1: Example of the use of port groups 'PowerSave' that denote ports that are required during a PowerSave mode. Not required communication resources could be deactivated during PowerSave mode.

10 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.

10.1 Calibration

AUTOSAR provides two mechanisms for calibration:

- **Port-based calibration** (based on the Parameter Software Components): 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.

10.1.1 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 parameter interface.
The components that contain the actual values of the calibration parameters are called “parameter software components”. In contrast to normal software-components, parameter software 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 parameter interface. Note that the parameter interface as well as the parameter software components are also used for fixed data exchange and not just used for calibration. The “implementation policy” of the elements on the port interface determines if it is fixed, const or variable data that is being accessed from the parameter software component.
The fact that a component is calibrated by a specific parameter software component is expressed through a connector between the corresponding ports. The calibration data is made available via the provide port of the parameter software component to a corresponding require port of any software component (compatibility rules do apply).
Since in this model the parameters are visible on the virtual bus, parameter software 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 sections.

10.1.1.1 Pure single instantiation

Figure 10.1 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 parameter software component.

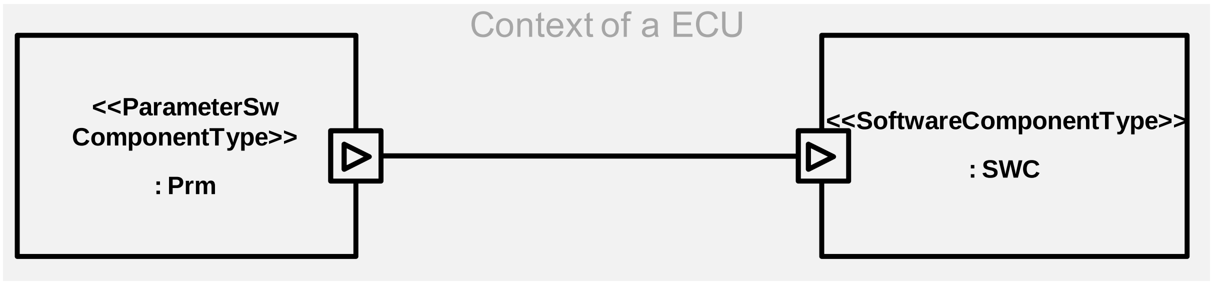


Figure 10.1 A software component has access to a calibration parameter encapsulated in a parameter software component

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

10.1.1.2 Multiple instantiation of the involved software components

Figure 10.2 and Figure 10.3 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.

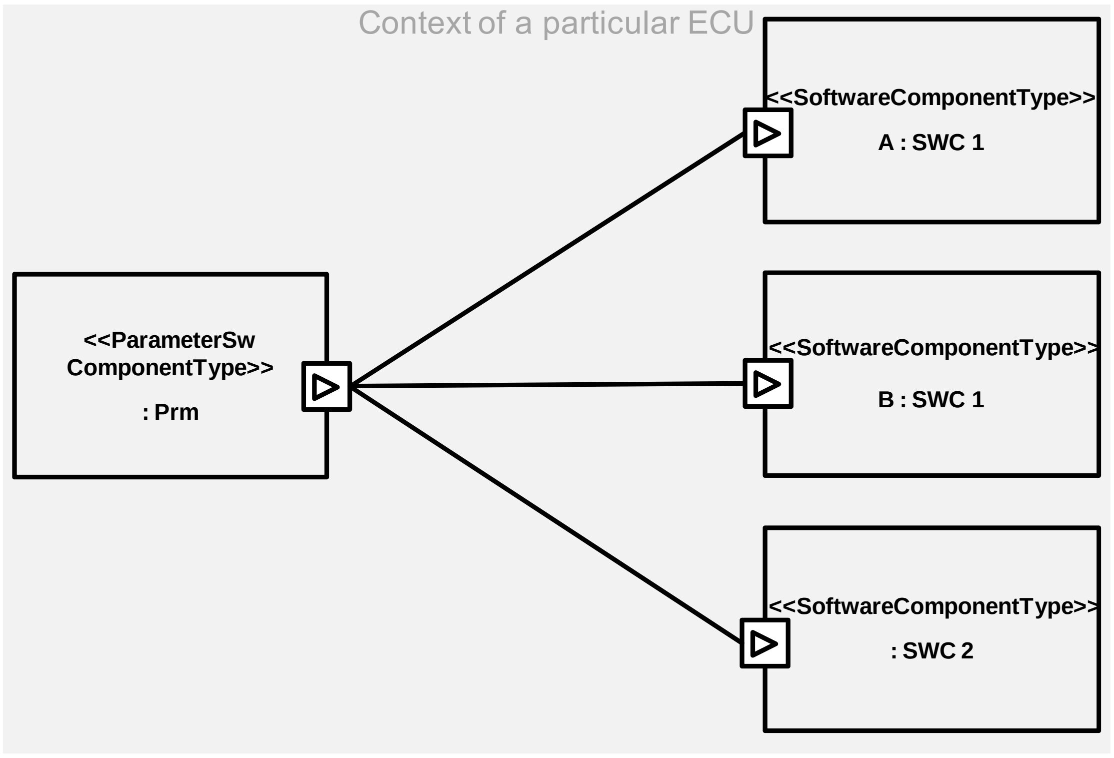

Context of a particular ECU
<<ParameterSw ComponentType>>
: Prm
<<SoftwareComponentType>>
A : SWC 1
<<SoftwareComponentType>>
B : SWC 1
<<SoftwareComponentType>>
: SWC 2

Figure 10.2 Two software components of the same type access the same calibration parameter encapsulated in a parameter software component

Since the (calibration) parameters need to reside on the same ECU as the software component accessing them, the parameter software component needs to be duplicated if the different software component instances are mapped onto different ECUs (see Figure 10.3).

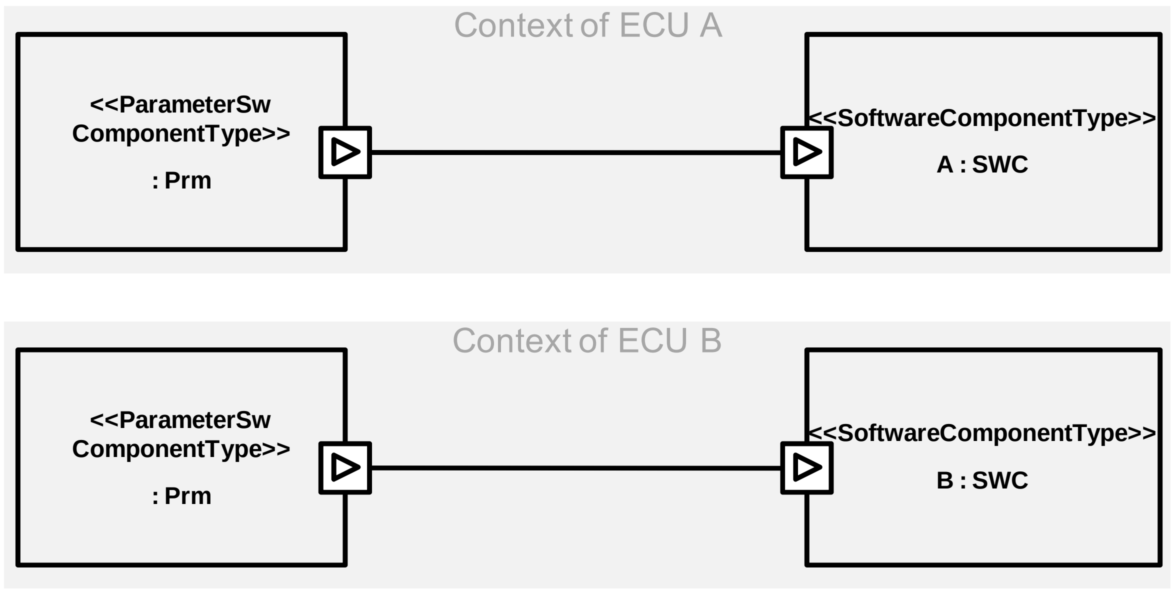

Context of ECU A
<<ParameterSw ComponentType>>
: Prm
<<SoftwareComponentType>>
A : SWC
Context of ECU B
<<ParameterSw ComponentType>>
: Prm
<<SoftwareComponentType>>
B : SWC

Figure 10.3 Like in Figure 10.1, but the software components are mapped onto different ECUs

10.1.1.3 Multiple instantiation of the involved calibration components

Figure 10.4 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.

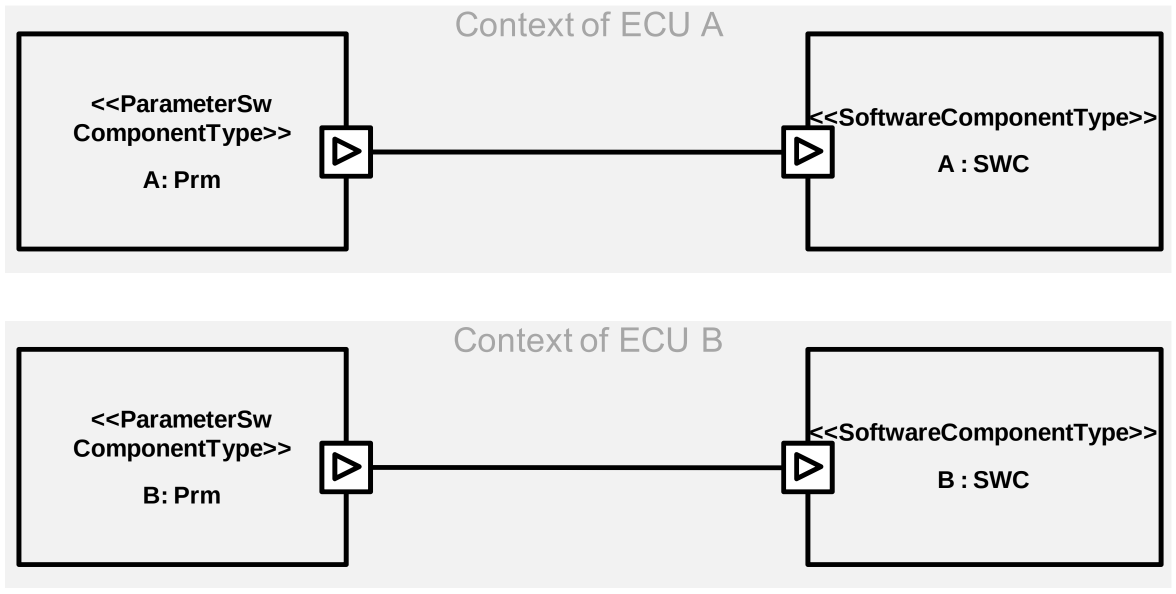

Context of ECU A
<<ParameterSw ComponentType>>
A: Prm
<<SoftwareComponentType>>
A : SWC
Context of ECU B
<<ParameterSw ComponentType>>
B: Prm
<<SoftwareComponentType>>
B : SWC

Figure 10.4 Two software components of the same type have been assigned different instances of the same Parameter Software Component Type.

10.1.2 Private calibration

The private calibration mechanism is based on parameters that are private and internal to a software component. From the software component implementation point of view a calibration parameter is a variable with only read-access during the normal operation of the ECU. A calibration parameter can be defined per instance of a software component (perInstanceParameter) or can be shared between all instances of a software component (sharedParameter).

Calibration parameters are not visible per se on the virtual functional bus, since it is considered an element associated to an internal behaviour of a software component.

Unlike the structure of software components and compositions which is considered to be specified during system design, the internal behaviour can be defined later in time when particular software components are supplied. With this respect the visibility of the private calibration parameters is rather a function of time, depending on who assigns them when.

10.2 Measurement

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

Communication between AUTOSAR SW-Components:
- VariableDataPrototypes enclosed in a sender-receiver interface
- Arguments of ClientServerOperations 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.

11 Interaction with Non-AUTOSAR-ECUs

11.1 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.:
- o Integration of an AUTOSAR ECU into an already existing system of ECUs
- o Connect system of AUTOSAR ECUs to already existing system of ECUs
- o 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.:
- o 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.:
- o 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.

11.2 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
  - o Unconnected ports of AUTOSAR-SW-Components
  - o 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
  - o services/protocols have to be managed and implemented in any case by master node (in this case AUTOSAR ECU)
  - o 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, ...)

Whether the non-AUTOSAR ECUs 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.

11.3 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 11.1: 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. Since the converter is here 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.

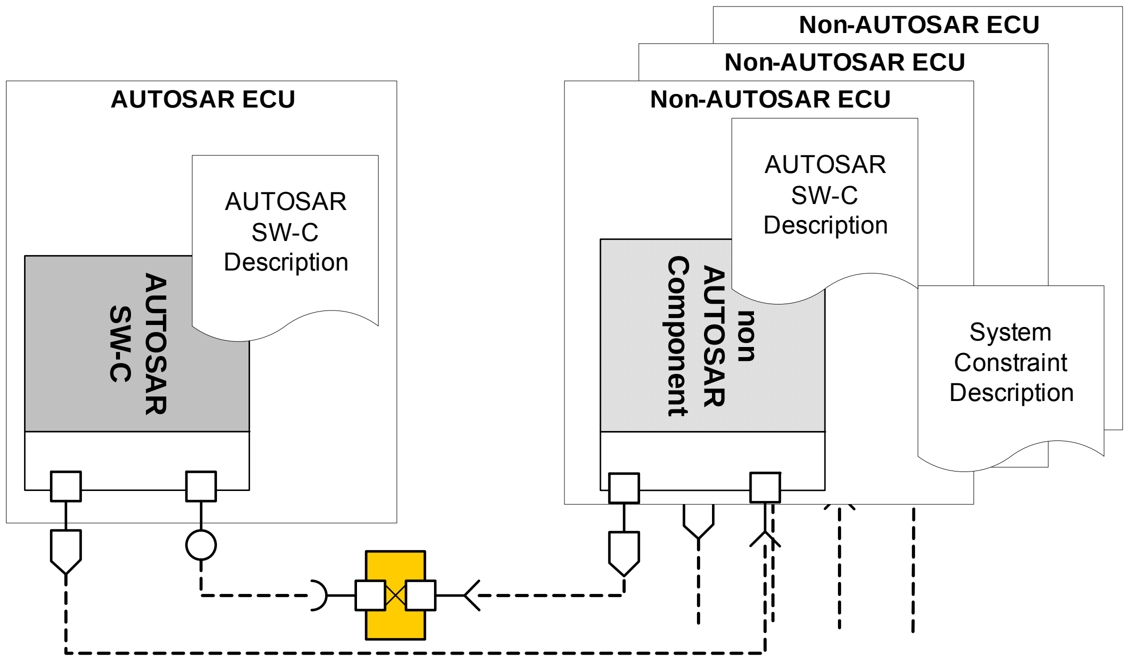


Figure 11.1: Interaction with non-AUTOSAR ECUs

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).

12 References

[1] Methodology
AUTOSAR_MOD_Methodology.pdf

[2] Glossary
AUTOSAR_TR_Glossary.pdf

[3] Main Requirements
AUTOSAR_RS_Main.pdf

[4] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList.pdf

[5] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[6] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[7] Specification of RTE
AUTOSAR_SWS_RTE.pdf

[8] Specification of Timing Extensions
AUTOSAR_TPS_TimingExtensions.pdf