Document Title	Specification of I/O Hardware Abstraction
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	47

Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R24-11

Document Change History

Date	Release	Changed by	Description
2024-11-27	R24-11	AUTOSAR Release Management	• Editorial changes
2023-11-23	R23-11	AUTOSAR Release Management	• No content changes
2022-11-24	R22-11	AUTOSAR Release Management	• Changed [SWS_IoHwAb_00145] to [SWS_IoHwAb_NA_00145]
2021-11-25	R21-11	AUTOSAR Release Management	• No content changes
2020-11-30	R20-11	AUTOSAR Release Management	• No content changes
2019-11-28	R19-11	AUTOSAR Release Management	• EcuAbstractionComponentType changed to EcuAbstractionSwComponentType • Changed Document Status from Final to published
2018-10-31	4.4.0	AUTOSAR Release Management	• Debugging section removed
2016-11-30	4.3.0	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation

△

2015-07-31	4.2.2	AUTOSAR Release Management	• Updated `IoHwAb_Init` function prototype
2014-10-31	4.2.1	AUTOSAR Release Management	• Editorial changes
2014-03-31	4.1.3	AUTOSAR Release Management	• Adapted the requirement format
2013-10-31	4.1.2	AUTOSAR Release Management	• Editorial changes • Removed chapter(s) on change documentation
2011-12-22	4.0.3	AUTOSAR Administration	• Update Version Check requirement
2010-09-30	3.1.5	AUTOSAR Administration	• Names of callback notification APIs have been corrected • Exported files <ModuleName>.h of underlying modules are used, instead of <ModuleName>_Types.h
2010-02-02	3.1.4	AUTOSAR Administration	• I/O Hardware Abstraction configuration has been removed from the EcucParamDef • Functional Diagnostics' interface has been added (DCM controls I/O signals) • Unnecessary classes, attributes and types removed • Legal disclaimer revised
2008-08-13	3.1.1	AUTOSAR Administration	• Legal disclaimer revised
2007-12-21	3.0.1	AUTOSAR Administration	• Auto generation of chapters 8 and 10 with the Metamodel • Update of tables and some chapters of the document to stay compliant with correlated documents • Document meta information extended
2007-01-24	2.1.15	AUTOSAR Administration	• Various images corrected in PDFversion (printing problems)

▽

△

2006-11-28	2.1.14	AUTOSAR Administration	• File structure updated • Traceability matrix corrected • Restriction for the usage of the SWC template • Chapter about IOHWAB Runnable concept reworked • Chapter about IOHWAB description reworked • Adjustments in the configuration chapter • Legal disclaimer revised • Release Notes added • "Advice for users" revised • "Revision Information" added
2006-05-16	2.0	AUTOSAR Administration	• Initial Release

Disclaimer

Contents

1 Introduction and functional overview

This specification specifies the functionality and the configuration of the AUTOSAR Basic Software I/O Hardware Abstraction. The I/O Hardware Abstraction is part of the ECU Abstraction Layer.

The I/O Hardware Abstraction shall not be considered as a single module, as it can be implemented as more than one module. This specification for the I/O Hardware Abstraction is not intended to standardize this module or group of modules. Instead, it is intended to be a guideline for the implementation of its functional interfaces with other modules.

Aim of the I/O Hardware Abstraction is to provide access to MCAL drivers by mapping I/O Hardware Abstraction ports to ECU signals. The data provided to the software component is completely abstracted from the physical layer values. Therefore, the software component designer does not need detailed knowledge about the MCAL driver's API and the units of the physical layer values anymore.

The I/O Hardware Abstraction is always an ECU specific implementation, because the requirements of the software components to the basic software have to be fitted to the features of a certain MCAL implementation.

The I/O Hardware Abstraction shall provide the service for initializing the whole I/O Hardware Abstraction.

The intention of this document is:

- to determine which part of the Software Component template shall be used when defining an I/O Hardware Abstraction.
- to explain the way to define generic ports, where ECU signals are mapped.

The intention of this document is not:

- to provide C-APIs
- to provide a specific formalization for every ECU signal, like it is done via the standardization of functional data (body domain, powertrain, chassis domain)

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the IOHWAB module that are not included in the [1, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
AUTOSAR	AUTomotive Open System ARchitecture
API	Application Programming Interface
BSW	Basic SoftWare
BSWMD	Basic SoftWare Module Description
C/S	Client/Server
DET	Default Error Tracer
ECU	Electronic Control Unit
HW	HardWare
IoHwAb	Input/Output Hardware Abstraction
ISR	Interrupt Service Routine
MCAL	MicroController Abstraction Layer
OS	Operating System
RTE	RunTime Environment
S/R	Sender/Receiver
SW	SoftWare
SWC	SoftWare Component (see [2] for further information)
XML	eXtensible Markup Language

Table 2.1: Acronyms and abbreviations used in the scope of this Document

Expression	Description	Example
Callback	Within this document, the term 'callback' is used for API services, which are intended for notifications to other BSW modules.	
Callout	Callouts are function stubs, which can be filled at configuration time, with the purpose to add functionality to the module that provides the callout.	
Class	A class represents a set of signals that has similar electrical characteristics.	Analogue class, Discrete class, ...
Client / Server communication	This definition is an extract from [3]: Client-server communication involves two entities, the client which is the requirer (or user) of a service and the server that provides the service. The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server, in the form of the RTE, waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client's request. So, the direction of initiation is used to categorize whether an AUTOSAR Software Component is a client or a server.	
Electrical Signal	An electrical signal is the physical signal on the pin of the ECU.	Physical input voltage at an ECU-Pin

▽

△

Expression	Description	Example
ECU pin	An ECU pin is an electrical hardware connection of the ECU with the rest of the electronic system.	
ECU Signal	An ECU Signal is the software representation of an electrical signal. An ECU signal has attributes and a symbolic name	Input voltage ,Discrete Output, PWM Input
ECU Signal Group	An ECU Signal Group is the software representation of a group of electrical signals.	
Attributes	Characteristics that can be Software (SW) and Hardware (HW) for each kind of ECU signals existing in an ECU. Some of the Attributes are fixed by the port definitions, others can be configured in the I/O Hardware Abstraction.	Range, Lifetime / delay
Sender-receiver communication	This definition is an extract from [3]: Sender-receiver communication involves the transmission and reception of signals consisting of atomic data elements that are sent by one component and received by one or more components. A sender-receiver interface can contain multiple data elements. Sender-receiver communication is one-way - any reply sent by the receiver is sent as a separate sender-receiver communication. A port of a component that requires an AUTOSAR sender-receiver interface can read the data elements described in the interface and a port that provides the interface can write the data elements.	
Symbolic name	The symbolic name of a ECU signal is used by the I/O Hardware Abstraction to make a link (function, pin)	

Table 2.2: Expressions used in this document

Expression	Description	Example
Range	This is a functional range and not an electrical range. All the range is used either for functional needs or for diagnosis detections For analogue ECU signals [lowerLimit...upperLimit] (Voltage, current). For the particular case of a resistance signal and a timing signal (period), the lowerLimit value can not be negative.	[-12Volts...+12Volts] (voltage) [0,1] (discrete signals) [0...upperLimit] (period timing signal) [-100...100%] (Duty Cycle based timing signal)
Resolution	This attribute is for many Classes dependent on the range and the Data Type. Example: (upperLimit - lowerLimit) / ($2^{datatypelength}$ -1) For the others classes, it is known and defined.	[-12 Volts...+12Volts] Data Type : 16 bits Resolution => 24 / 65535
Accuracy	It depends of hardware peripheral used for acquisition and/or generation.	ADC converter could be a 8/10/12/16 bits converter
Inversion	Inversion between the physical value and the logical value. This attribute is not visible but done by I/O Hardware Abstraction to deliver expected values to users.	Physical HighState (signal=False) Physical LowState (signal=True)
Sampling rate	Time period required to get a signal value.	Sampling rate for a sampling windows (burst)

Table 2.3: ECU signal attributes

3 Related documentation

3.1 Input documents & related standards and norms

[1] Glossary
AUTOSAR_FO_TR_Glossary

[2] Software Component Template
AUTOSAR_CP_TPS_SoftwareComponentTemplate

[3] Specification of RTE Software
AUTOSAR_CP_SWS_RTE

[4] General Specification of Basic Software Modules
AUTOSAR_CP_SWS_BSWGeneral

[5] Layered Software Architecture
AUTOSAR_CP_EXP_LayeredSoftwareArchitecture

[6] Requirements on I/O Hardware Abstraction
AUTOSAR_CP_RS_IOHWAbstraction

[7] General Requirements on SPAL
AUTOSAR_CP_RS_SPALGeneral

[8] Specification of ECU Resource Template
AUTOSAR_CP_TPS_ECUResourceTemplate

[9] Specification of ECU State Manager
AUTOSAR_CP_SWS_ECUStateManager

[10] Specification of ADC Driver
AUTOSAR_CP_SWS_ADCDriver

[11] Specification of DIO Driver
AUTOSAR_CP_SWS_DIODriver

[12] Specification of ICU Driver
AUTOSAR_CP_SWS_ICUDriver

[13] Specification of PWM Driver
AUTOSAR_CP_SWS_PWMDriver

[14] Specification of Port Driver
AUTOSAR_CP_SWS_PortDriver

[15] Specification of GPT Driver
AUTOSAR_CP_SWS_GPTDriver

[16] Specification of SPI Handler/Driver
AUTOSAR_CP_SWS_SPIHandlerDriver

[17] Specification of OCU Driver

AUTOSAR_CP_SWS_OCUDriver

3.2 Related specification

AUTOSAR provides a General Specification on Basic Software modules [4, CP SWS BSW General], which is also valid for IO Hardware Abstraction.

Thus, the specification SWS BSW General shall be considered as additional and required specification for IO Hardware Abstraction.

4 Constraints and assumptions

4.1 Limitations

No limitations

4.2 Applicability to car domains

No restrictions

5 Dependencies to other modules

5.1 Interface with MCAL drivers

5.1.1 Overview

The following picture shows the I/O Hardware Abstraction. It is located above MCAL drivers. That means the I/O Hardware Abstraction will call the driver's APIs for managing on chip devices. The configuration of the MCAL drivers depends on the quality of the ECU signals that is required by the SWCs. For instance, it could be necessary to have notifications when a relevant change occurs on the pin level (rising edge, falling edge). The system designer has to configure the MCAL drivers to allow notifications for a given signal. Notifications are generated by MCAL drivers and are handled within the I/O Hardware Abstraction.

Please notice that I/O Hardware Abstraction is not intended to abstract GPT functionalities, but rather to use them to perform its own functionalities. The interfacing with GPT driver is shown because it is part of the MCAL.

The following picture shows all interfaces with MCAL drivers:

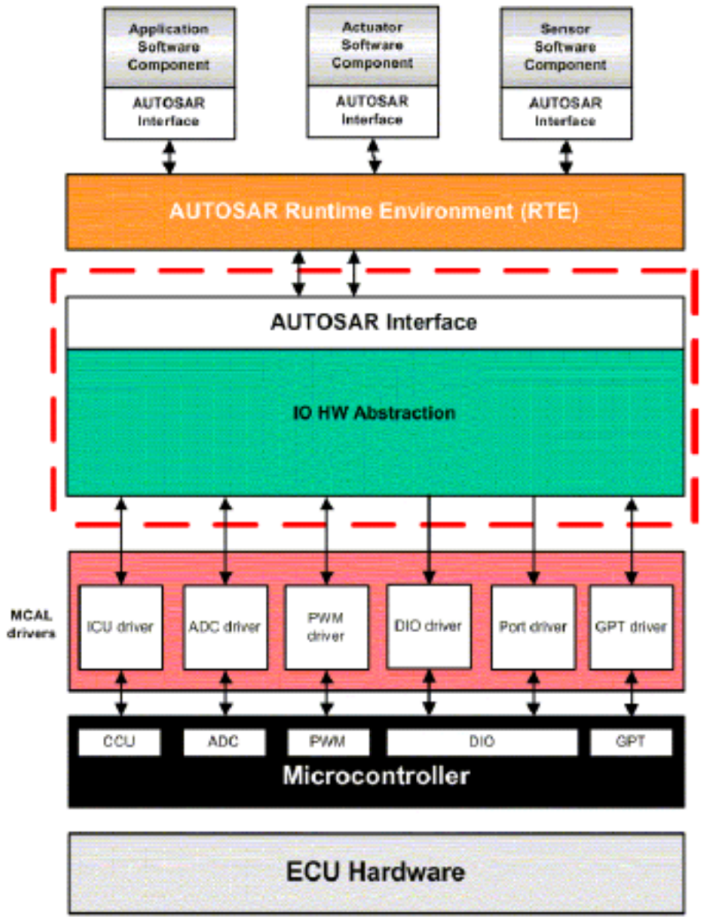

Figure 5.1: Interfaces with MCAL drivers

5.1.2 Summary of interfaces with MCAL drivers

[SWS_IoHwAb_00078]

Upstream requirements: SRS_BSW_00384

⌈The I/O Hardware Abstraction implementation shall provide Software Components with access to all MCAL drivers.⌋

		MCAL drivers					
IoHwAb	**ADC driver**	**OCU driver**	**PWM driver**	**ICU driver**	**DIO driver**	**PORT driver**	**GPT driver**
Calls API of	X	X	X	X	X	X	X
Receives notifications from	X	X	X	X	-	-	X

The table above must be read as following:

- The I/O Hardware Abstraction calls API of the ADC driver
- The I/O Hardware Abstraction receives notifications from the ADC driver.
- The I/O Hardware Abstraction does not receive notifications from the DIO driver.

A complete list of all APIs is given in chapter 8.8.

5.2 Interface with the communication drivers

[SWS_IoHwAb_00079]

Upstream requirements: SRS_BSW_00384, SRS_IoHwAb_12242

⌈The I/O Hardware Abstraction implementation shall provide Software Components with access to communication drivers (for instance by SPI), if on-board devices are managed.⌋

The following picture shows the I/O Hardware Abstraction, where some signals come from / are set via the SPI handler / driver.

According to the Layered Software Architecture [5] (ID03-16), the I/O Hardware Abstraction contains dedicated drivers to manage external devices for instance:

- A driver for external ADC driver, connected via SPI.
- A driver for external I/O realized on an ASIC device, connected via SPI.

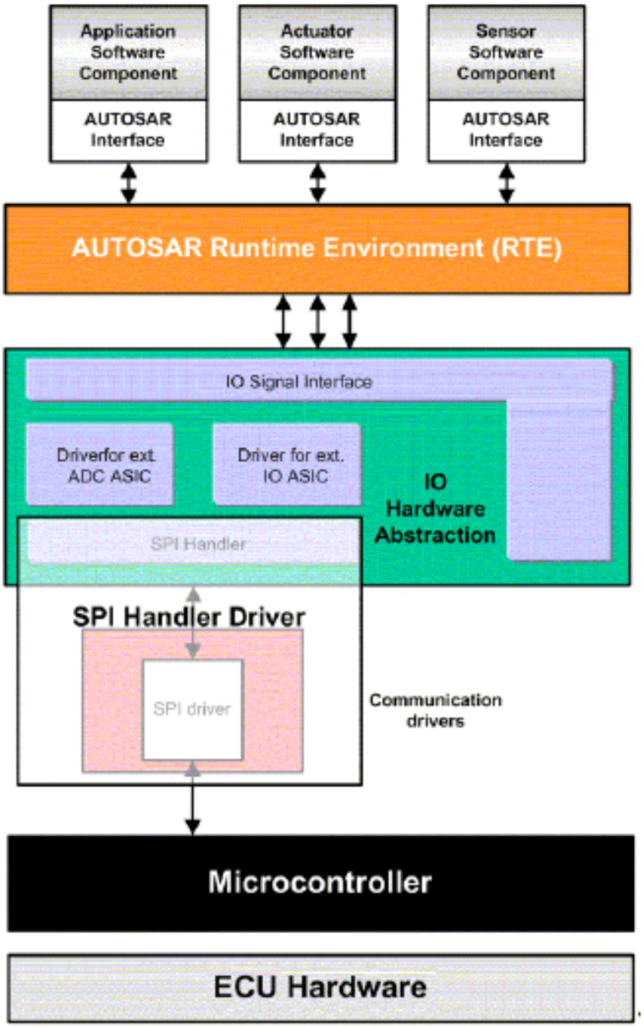

Figure 5.2: Interfaces with communication drivers

5.3 Interface with System Services

[SWS_IoHwAb_00044]

Upstream requirements: SRS_BSW_00336, SRS_BSW_00384, SRS_BSW_00101

⌈The I/O Hardware Abstraction implementation shall interface with the following system services:

- ECU State Manager (init function)
- DET: Default Error Tracer
- BSW Scheduler

⌋

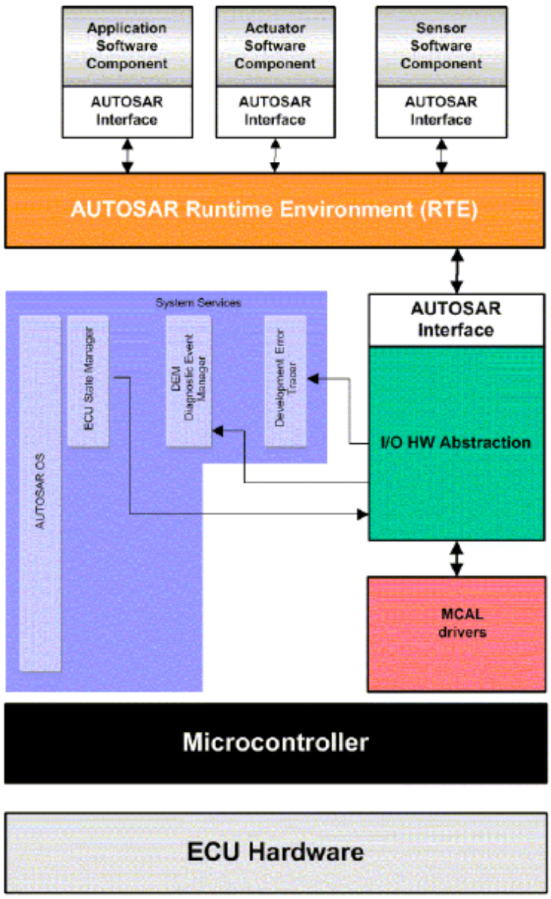

Figure 5.3: Interfaces with system services

5.4 Interface with DCM

The I/O Hardware Abstraction shall provide interfaces to DCM, for functional diagnostics of the software components. DCM will use functional diagnostics for reading and controlling the implemented ECU signals.

The prototypes of the interfaces provided to DCM shall be within a header file IoHwAb_Dcm.h.

For details of the interfaces, refer section 8.6.

5.5 File structure

5.5.1 Code file structure

[SWS_IoHwAb_00097] ⌈The code file structure shall not be defined within this specification.⌋

5.5.2 Header file structure

As there can be multiple, project-specific instances of the I/O Hardware Abstraction, the file structure cannot be specified.

[SWS_IoHwAb_00112] ⌈File names should be prefixed with 'IoHwAb_<ComponentName>_<reference>' (where the field <reference> can be an implementation-specific category and the field <ComponentName> is the name of the atomic software component, i.e. the instance of the I/O Hardware Abstraction) in order to avoid name clashes.⌋

6 Requirements Tracing

The following tables reference the requirements specified in [4], [6], [7] and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[SRS_BSW_00101]	The Basic Software Module shall be able to initialize variables and hardware in a separate initialization function	[SWS_IoHwAb_00036] [SWS_IoHwAb_00044] [SWS_IoHwAb_00059] [SWS_IoHwAb_00060] [SWS_IoHwAb_00061]
[SRS_BSW_00333]	For each callback function it shall be specified if it is called from interrupt context or not	[SWS_IoHwAb_00033]
[SRS_BSW_00336]	Basic SW module shall be able to shutdown	[SWS_IoHwAb_00036] [SWS_IoHwAb_00044]
[SRS_BSW_00384]	The Basic Software Module specifications shall specify at least in the description which other modules they require	[SWS_IoHwAb_00044] [SWS_IoHwAb_00078] [SWS_IoHwAb_00079]
[SRS_BSW_00414]	Init functions shall have a pointer to a configuration structure as single parameter	[SWS_IoHwAb_00157] [SWS_IoHwAb_00158]
[SRS_BSW_00423]	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	[SWS_IoHwAb_00001]
[SRS_BSW_00440]	The callback function invocation by the BSW module shall follow the signature provided by RTE to invoke servers via `Rte_Call` API	[SWS_IoHwAb_00143]
[SRS_BSW_00441]	Naming convention for type, macro and function	[SWS_IoHwAb_00102]
[SRS_BSW_00450]	A Main function of a un-initialized module shall return immediately	[SWS_IoHwAb_00035]
[SRS_IoHwAb_00002]	The I/O Hardware Abstraction shall provide an interface to the DCM that allows to control and read the configured signals	[SWS_IoHwAb_00135] [SWS_IoHwAb_00136] [SWS_IoHwAb_00138] [SWS_IoHwAb_00139] [SWS_IoHwAb_00140] [SWS_IoHwAb_00142]
[SRS_IoHwAb_12242]	The IO Hardware Abstraction shall hide any communication over ECU internal onboard peripherals to access Signals	[SWS_IoHwAb_00079]
[SRS_IoHwAb_12248]	The IO Hardware Abstraction module shall keep the ECU hardware safe	[SWS_IoHwAb_00038]
[SRS_IoHwAb_12451]	The IO Hardware Abstraction module shall not decide on its own to switch an output on again that has been switched off for hardware protection reasons	[SWS_IoHwAb_00039]
[SRS_SPAL_12056]	All driver modules shall allow the static configuration of notification mechanism	[SWS_IoHwAb_00032] [SWS_IoHwAb_00033] [SWS_IoHwAb_00034]

Table 6.1: Requirements Tracing

7 Functional specification

7.1 Integration code

The I/O Hardware Abstraction, as a part of the ECU abstraction, has been defined as **integration code**.

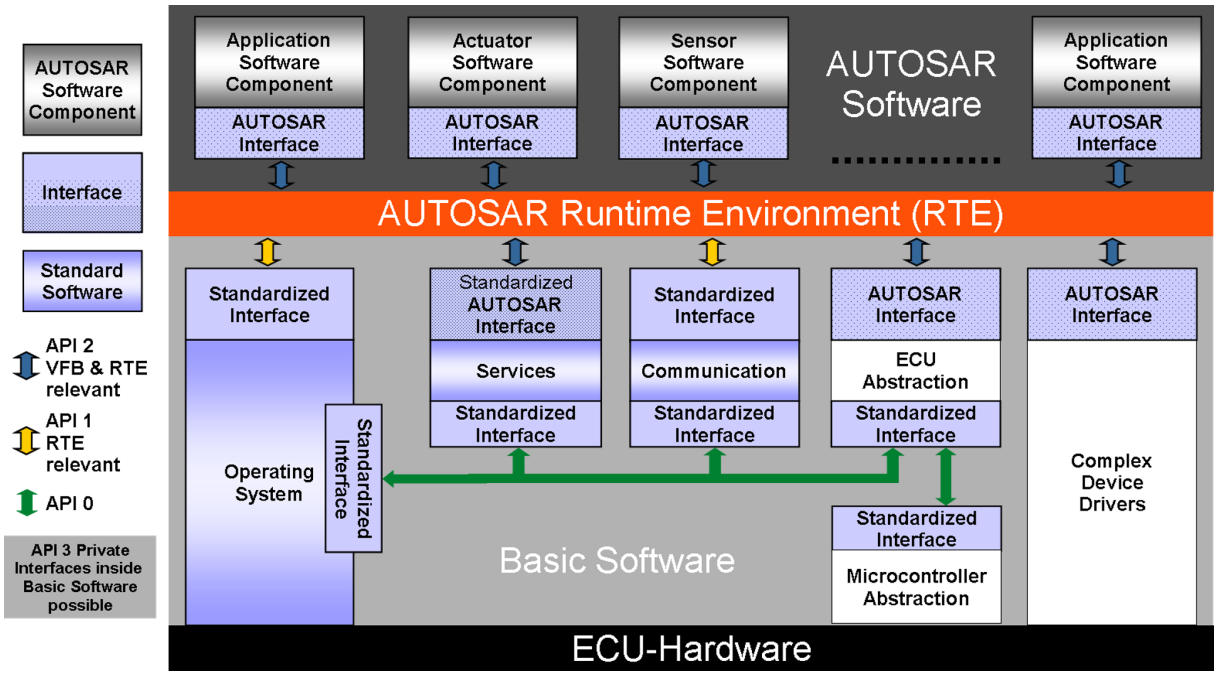

Figure 7.1: AUTOSAR architecture

7.1.1 Background & Rationale

According to the AUTOSAR glossary [1], integration code is ECU schematic dependent software located below the AUTOSAR RTE.

7.1.2 Requirements for integration code implementation

The following requirements for the I/O Hardware Abstraction are related to hardware protection.

[SWS_IoHwAb_00038]

Upstream requirements: SRS_IoHwAb_12248

⌈Integration code usually means that this software is designed to suite a specific ECU hardware layout. All strategies to protect the hardware shall be included in this soft-

ware. This document does not intend to standardize or give a recommendation for such hardware protection.⌋

Hardware protection means, that the I/O Hardware Abstraction is able to cut off an output signal, when a failure (short circuit to ground/power supply, over temperature, overload ...) is detected on the certain output.

[SWS_IoHwAb_00039]

Upstream requirements: SRS_IoHwAb_12451

⌈The I/O Hardware Abstraction shall not contain strategies for failure recovery. Failure recovery actions can only be decided by the responsible SWC.⌋

The internal behavior of the I/O Hardware Abstraction is project-specific and cannot be standardized.

There is no I/O Hardware Abstraction scalability. The SWC specifies what is needed (quality of signal) and the I/O Hardware Abstraction has to provide it.

7.2 ECU Signals Concept

7.2.1 Background & Rationale

The I/O Hardware Abstraction cannot provide Standardized AUTOSAR Interfaces to AUTOSAR SW-Cs, as its interfaces to the upper layer strongly depend on the chain of signal acquisition. Instead, the I/O Hardware Abstraction provides AUTOSAR Interfaces.

These AUTOSAR Interfaces represent an abstraction of electrical signals coming from the ECU inputs / addressed to ECU outputs.

Alternatively, these electrical signals may also come from other ECUs or be addressed to other ECUs (e.g. via a CAN network).

Ports are entry points of AUTOSAR components. They are typified by an AUTOSAR interface. These interfaces correspond to "ECU signals".

The concept of ECU signals comes from the necessity to guarantee the interchangeability of hardware platforms.

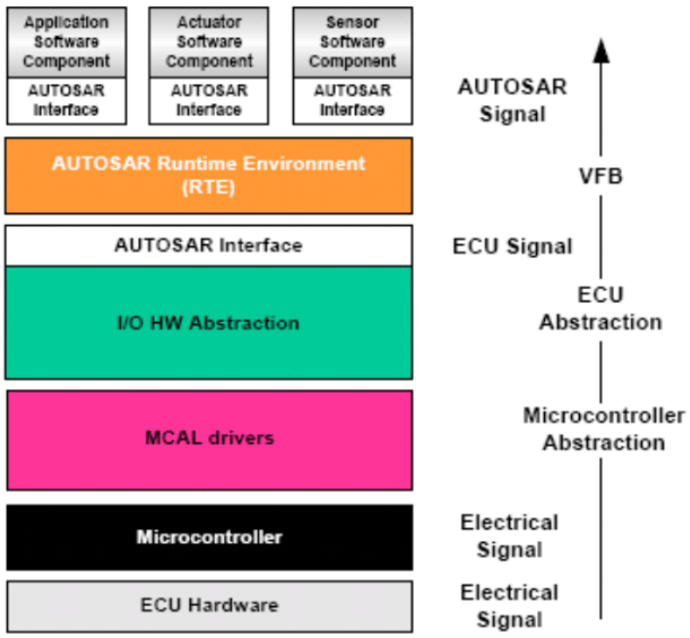

Figure 7.2: ECU signals

7.2.2 Requirements about ECU signals

The I/O Hardware Abstraction handles all inputs and outputs directly connected to the ECU (except those that have a dedicated driver, like CAN, see requirement [SWS_IoHwAb_00063]).

It includes all inputs and outputs, directly mapped to microcontroller ports, or to an onboard peripheral. All communication between the microcontroller and the peripherals (except sensors and actuators and peripherals managed by complex drivers) are hidden by the I/O Hardware Abstraction, while considering the provided interfaces.

An ECU is connected to the rest of the system through networks and inputs and output pins. Networks are out of scope of this document.

[SWS_IoHwAb_00063] ⌈An ECU signal represents one electrical signal, which means at least one input or output ECU pin.⌋

The software in this layer shall abstract the ECU pins. Looking from this place (for example using an oscilloscope) inputs and outputs are only electrical signals. Hence, all that is defined in this document is related to this concept of electrical signals. One extension of this concept is diagnosis (electrical failure status). Diagnosis is not visible from ECU connectors but is provided by the I/O Hardware Abstraction.

Electrical signals with similar behavior may form a class. Therefore, ECU signals, which denote the software representation of electrical signals may have an association to an implementation-specific class.

7.3 Attributes

7.3.1 Background & Rationale

Even though most of the characteristics of each ECU Signal are defined by the SWC, some properties have to be added to each signal to provide the signal quality the SWC expects.

7.3.2 Requirements about ECU signal attributes

To detail the chain of signal-acquisition, a list of Attributes is defined to identify configurable characteristics of ECU signals.

7.3.2.1 Filtering/Debouncing Attribute

[SWS_IoHwAb_00019] ⌈All ECU Signals shall have a Filtering/Debounce Attribute, so that the captured 'raw' - values can be filtered or debounced before passing them to the upper layer. This attribute is only reasonable for input signals. It influences the implementation of acquisition and access to the signal values.⌋

7.3.2.2 Age Attribute

All ECU signals handled by I/O Hardware Abstraction depend on the ECU hardware design. This means that the time to set ECU Output signals and the time to get ECU Input signals could be different from one to other ECU signal. So to guarantee a template behavior for all kind of ECU signals (Input / Output) a common Age Attribute is defined and it shall be configured for each ECU signal.

[SWS_IoHwAb_00021] ⌈All ECU signals shall have an Age Attribute. The Age Attribute has two specific names according to the direction of ECU signal (Input / Output). Anyway, it always contains a maximum time value. Following descriptions explain the meaning of this Attribute for each kind of ECU signals.

- ECU Input signals: the specific functionality of this attribute is to limit the signals lifetime. The value defines the maximum allowed age for data of this signal. If the

lifetime is 0, the signal has to be retrieved from the physical register, immediately. If the lifetime is greater than 0, the signal is valid for the specified time.

- ECU Output signals: the specific functionality of this attribute is to limit the signal output to a maximum delay. The value defines the maximum allowed time until this signal is actually set. If delay is 0, then the signal has to be set to the physical register, immediately. If the delay is greater than 0, the signal can be set until the configured time has elapsed.

⌋

7.4 I/O Hardware Abstraction and Software Component Template

Note about this chapter: This chapter refers to document [8].

Changes inside this document may influence the content of this chapter.

7.4.1 Background & Rationale

This approach allows defining the standardization deepness. As explained previously, the implementation is integration code. Therefore, this chapter only summarizes how to define the I/O Hardware Abstraction as a Software Component (SWC), and gives a short overview of the internal behavior. The internal behavior description mainly covers BSW scheduling mechanisms.

7.4.2 Requirements about the usage of Software Component template

[SWS_IoHwAb_00001]

Upstream requirements: SRS_BSW_00423

⌈The I/O Hardware Abstraction shall be based upon the Software Component Template as specified in document [8].⌋

In the same manner as in any other Software Component, the I/O Hardware Abstraction might be sub-structured, depending on the complexity of an ECU.

Indeed, the I/O Hardware Abstraction is a classical Component Prototype, that can be atomic or composed and that provides and requires interfaces. Moreover, I/O Hardware Abstraction may only interact by means of their PortPrototypes with other Software Components above the RTE. Hidden dependencies that are not expressed by means of PortPrototypes are not allowed.

However, the I/O Hardware Abstraction interfaces on one side the MCAL drivers via Standardized Interfaces and on the other side the RTE. Hence, I/O Hardware Abstraction shall respect the virtual ports concept.

[SWS_IoHwAb_00025] ⌈The I/O Hardware Abstraction shall be implemented as one or more instances of the EcuAbstractionSwComponentType.⌋

See [8] for further information about the EcuAbstractionSwComponentType.

An instantiation of EcuAbstractionSwComponentType provides a set of ports. During RTE Generation, only those that are connected with Software Components are taken into account.

This chapter gives an overview of the virtual ports concept and runnable entities applied to the I/O Hardware Abstraction needs. The following chapters of this document describe the points set out here in more detail.

7.4.2.1 Ports concept and I/O Hardware Abstraction

This is an overview of recommendations for defining Ports of I/O Hardware Abstraction using the Software Component template.

Further chapters in this document go deeper in usage of ports for I/O Hardware Abstraction. Nevertheless, it is advised to read the Software Component Template document [8] to be aware of all terms and all concepts used.

The attributes described in chapter 7.3 shall be defined by annotating the ports of the I/O Hardware Abstraction components with an IoHwAbstractionServerAnnotation (see [8]).

7.4.2.2 Software Component and Runnable concept

Software Components have functions to realize their strategies and internal behaviors. These are partly described using runnable entities. The former is contained in runnables and the latter depends of runnables design. Runnable entities are provided by the Atomic Software Component and are (at least indirectly) a subject for scheduling by the underlying operating system.

An implementation of an atomic Software Component has to provide an entry-point to code for each Runnable in its "InternalBehavior". For more information, please refer to the specification [8].

The runnable entities are the smallest code-fragments, which can be activated independently. They are provided by the Atomic Software Component and are activated by the RTE. Runnables are for instance set up to respond to data exchange or operation invocation on a server.

The runnable entities have three possible states: Suspended, Enabled and Running. During run-time, each runnable of an atomic Software Component is (by being a member of an OS task) in one of these states.

For a sight of available choices and attributes to define each runnables of the Atomic Software Component, please refer to specification [8].

7.5 Scheduling concept for I/O Hardware Abstraction

7.5.1 Background & Rationale

The I/O Hardware Abstraction may consist of several BSW modules (e.g. onboard device driver).

Each of these BSW modules can provide BSW runnable entities (also called BswModuleEntity in the RTE Specification (see [3]).

To make a parallel, a BswModuleEntity is the equivalent of SWC runnable entities, for which the AUTOSAR glossary [1] gives the following definition: "A Runnable Entity is a part of an Atomic Software-Component (definition) which can be executed and scheduled independently from the other Runnable Entities of this Atomic Software-Component".

This means that the I/O Hardware Abstraction can use Runnable Scheduling and BSW Scheduling simultaneously. The Runnable Scheduling handles the Runnable Entities and is mandatory. Unlike the Runnable Scheduling, the BSW Scheduling is optional and the interfacing with the BSW Scheduler has to be done manually.

In case of SWC runnable entities, these are called in AUTOSAR OS Tasks bodies. Runnables are given in the SWC description. Activation of SWC runnables strongly depends on RTE events.

In the same way than SWCs are most often activated by RTEEvents, the schedulables BswModuleEntities can be activated by BswEvents. There is also a kind of BswModuleEntity which can be activated in interrupt context. This leads to two sub-classes: BswSchedulableEntity and BswInterruptEntity.

7.5.2 Requirements about I/O Hardware Abstraction Scheduling concept

7.5.2.1 Operations for interfaces provided by Ports

The I/O Hardware Abstraction, described from the interfaces point of view, implements the counterpart of the PortInterfaces defined by the SW-C, i.e. it provides Runnable Entities that implement the Provide Ports (Server port, Sender/Receiver port) required by the SW-C.

[SWS_IoHwAb_00068] ⌈The implementation behind the service of the I/O Hardware Abstraction's Provide Ports is ECU specific and the mapping to the corresponding "PortInterface" shall be documented in the Software Component description.⌋

7.5.2.1.1 Get operation

[SWS_IoHwAb_00069] ⌈For an ECU Signal associated with a PortInterface configured as an input signal, the I/O Hardware Abstraction shall provide a GET operation, and the operation short name can be freely choose.⌋

7.5.2.1.2 Set operation

[SWS_IoHwAb_00070] ⌈For an ECU Signal associated with a PortInterface configured as an output signal, the I/O Hardware Abstraction shall provide an SET operation, and the operation shortname can be freely choose.⌋

7.5.2.2 Notification and/or Callback

[SWS_IoHwAb_00032]

Upstream requirements: SRS_SPAL_12056

⌈The I/O Hardware Abstraction shall define BswInterruptEntities (a sub-class class of BswModuleEntity by opposition to BswSchedulableEntity) to fulfill notification and/or callback mechanisms to exchange data with other modules below the RTE within an interrupt context.⌋

The I/O Hardware Abstraction may contain one or several callback functions. The available callback functions need to be hooked up to the notification interfaces of the MCAL drivers. Therefore, they have to respect the prototype definition of the MCAL drivers (no passing parameter, no return parameter).

[SWS_IoHwAb_00033]

Upstream requirements: SRS_BSW_00333, SRS_SPAL_12056

⌈The implementation has to take into consideration, that the callback functions will be executed in interrupt context.⌋

Callback functions can additionally provide the capability to trigger Software Components outside of the I/O Hardware Abstraction. These notifications need to be handled through the RTE (sender port).

[SWS_IoHwAb_00034]

Upstream requirements: SRS_SPAL_12056

⌈The number of available callback functions and the order of execution will be implementation dependent and must be documented in the I/O Hardware Abstraction BSWMD.⌋

[SWS_IoHwAb_00143]

Upstream requirements: SRS_BSW_00440

⌈The function prototype for the callback function functions of the I/O Hardware Abstraction which are routed via RTE shall be implemented according to the following rule: StdReturnType Rte_Call_<p>_<o>(<parameters>)⌋

The callback functions have to be to be compatible to Rte_Call_<p>_<o> API of the RTE to enable a type safe configuration and implementation of AUTOSAR Services and IO Hardware Abstraction.

7.5.2.3 Main function / job processing function

[SWS_IoHwAb_00035]

Upstream requirements: SRS_BSW_00450

⌈The I/O Hardware Abstraction may contain one or several job processing functions that are BswSchedulableEntities (a sub-class of BswModuleEntity by opposition to BswInterruptEntity, e.g. one for each device driver). They shall be activated according to their use.

They will be time-triggered by the BSW Scheduler. They could be synchronized to the execution of the other runnable entities.

The number of BswSchedulableEntities and their order of execution will be implementation dependent and must be documented in the I/O Hardware Abstraction description.⌋

7.5.2.4 Initialization, De-initialization and/or Callout

[SWS_IoHwAb_00036]

Upstream requirements: SRS_BSW_00336, SRS_BSW_00101

⌈The I/O Hardware Abstraction shall define BswModuleEntries to exchange data with other software below the RTE outside interrupt context, for example in case of BSW initialization/de-initialization.⌋

These BswModuleEntries are linked to a dedicated BswModuleEntity, which will be called to perform the service / exchange the data.

The I/O Hardware Abstraction may contain one or several initialization and de-initialization functions (e.g. one for each device driver). Similar to the MCAL drivers the initialization functions shall contain a parameter to be able to pass different configurations to the device drivers. This function shall initialize all local and global variables used by the I/O Hardware Abstraction driver to an initial state.

[SWS_IoHwAb_00037] ⌈The initialization/de-initialization functions shall be used/handled by the ECU State Manager, exclusively. For more information, refer to [9].

The number of available functions and the order of execution are implementation-dependent and must be documented in the I/O Hardware Abstraction description.⌋

7.5.2.5 I/O Hardware Abstraction scheduling examples

7.5.2.5.1 Interface provided by ADC and I/O Hardware Abstraction

The following example shows a scheduling example for an ADC conversion.

The I/O Hardware Abstraction shall provide two P-ports.

The Software Component interface in this example is af_pressure.

The ECU state manager is able to trigger a BswModuleEntry for initialization of the ADC driver (Call of Adc_Init() with the Adc_ConfigType structure).

Use Case: The software component needs the af_pressure value.

1. RTE triggers the OP_GET operation of the dedicated P-Port.
2. R1 is a runnable entity and it allows to call the appropriated ADC driver services
   - ADC_EnableNotification
   - ADC_StartGroupConversion

3. At the end of conversion, the ADC triggers the BswModuleEntry R2, within interrupt context. This is possible since the notification is allowed for this interface. The ADC_NotificationGroup() function is specified in the ADC driver.
4. The notification is then "sent" to the Software Component via an RTEevent.

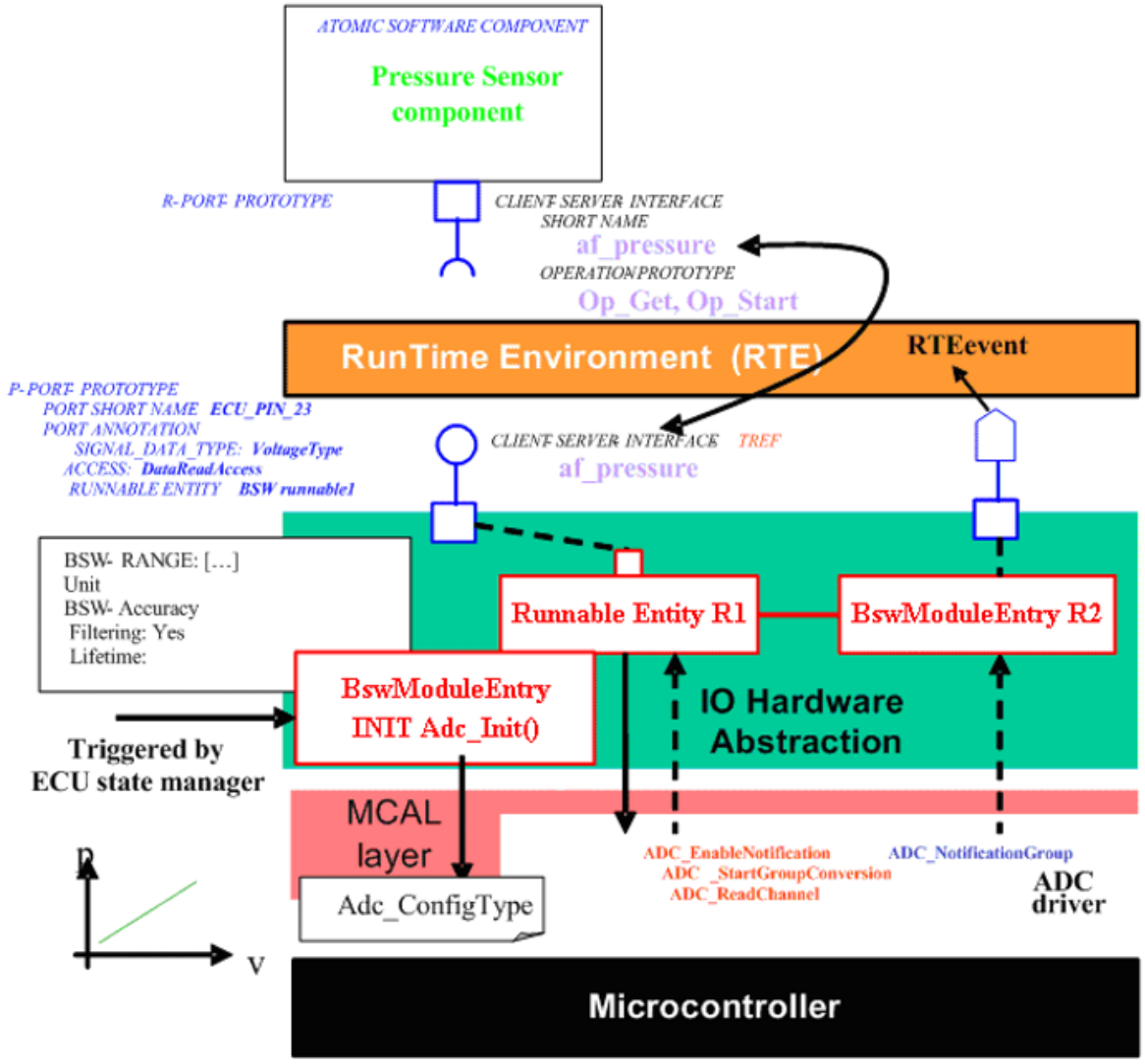

Figure 7.3: Example of IoHwAb runnables

The sequence diagram of this example is in chapter 9.

7.5.2.5.2 Synchronous scheduling with Runnable Entities and BswSchedulableEntities

The following example shows a scheduling example for setting a Lamp linked to a SMART power.

The SMART power is connected to the microcontroller by SPI bus. Hence, the dedicated piece of code uses the SPI Handler/Driver.

The FrontLeftLamp value to be set by the RTE is in an I/O Hardware Abstraction buffer.

An output line to another SMART power is set synchronously to trigger an ADC conversion of the same electrical signal by the ADC driver.

At the end of conversion, the converted result is available and the notification is set to the Analog input manager to store the value inside a buffer, available for diagnosis purpose.

In this example, the periodical treatment is realized by a BswSchedulableEntity.

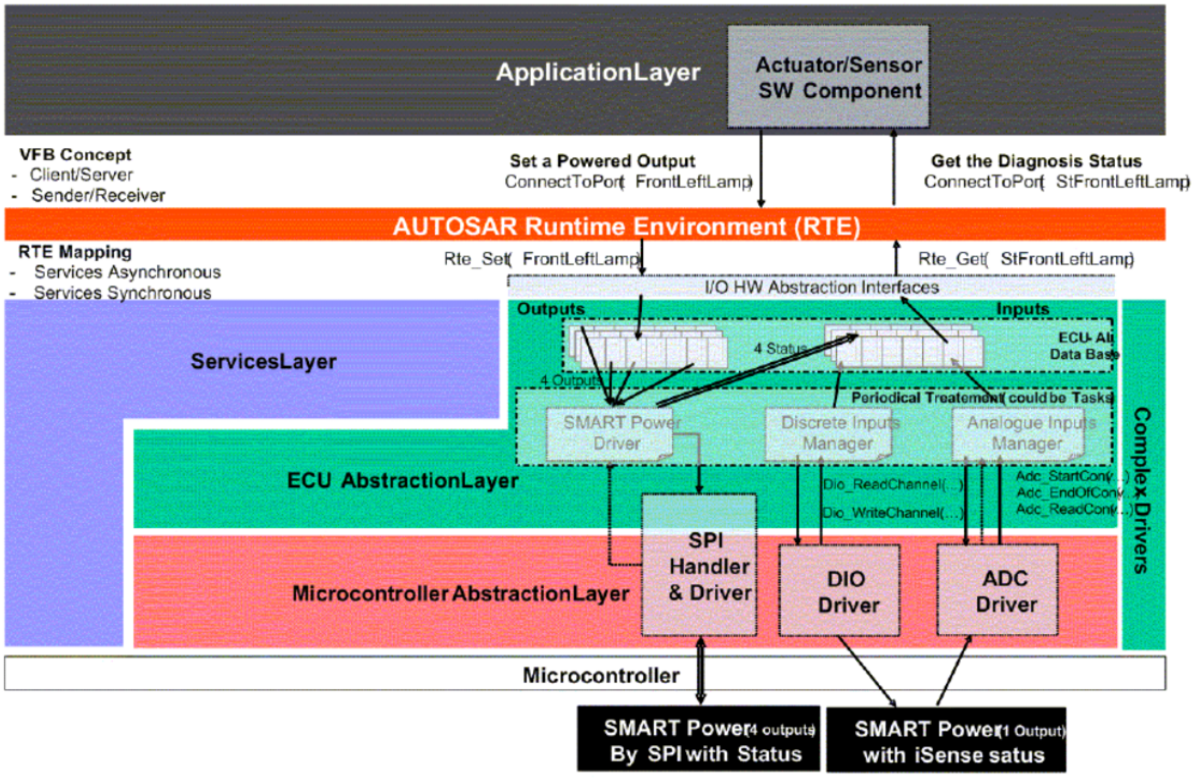

Figure 7.4: Example of IoHwAb runnable - cyclic setting of output and diagnosis

7.6 Error Classification

7.6.1 Development Errors

[SWS_IoHwAb_91001] Definiton of development errors in module IoHwAb ⌈

Type of error	*Related error code*	*Error value*
Up to the implementer to define error he wants to report	Up to the implementer	0x01

⌋

7.6.2 Runtime Errors

There are no runtime errors.

7.6.3 Production Errors

There are no production errors.

7.6.4 Extended Production Errors

Error Name:	Up to the implementer to define error he wants to report IOHWAB_E_<DESCRIPTIVE_NAME>[_<INSTANCE>]	
Short Description:	Up to the implementer	
Long Description:	Up to the implementer	
Detection Criteria:	Fail	Up to the implementer
	Pass	Up to the implementer
Secondary Parameters:	Up to the implementer	
Time Required:	Up to the implementer	
Monitor Frequency	Up to the implementer	

7.7 Other requirements

For details refer to the chapter 5.1.8 "Version Check" in CP_SWS_BSWGeneral [4].

7.8 I/O Hardware Abstraction layer description

7.8.1 Background & Rationale

The I/O Hardware Abstraction layer has some analogies with a Software Component, especially regarding port definition for communication through the RTE. The main difference is that the I/O Hardware Abstraction is below the RTE (in the ECU Abstraction Layer). The I/O Hardware Abstraction is a kind of interface between Basic Software modules and Application Software.

For the I/O Hardware Abstraction, but also for Services, the current methodology requires filling out two different templates. For example, in order to integrate an NVRAM Manager on an AUTOSAR ECU one would use the BSWMD to document its needs for the BSW Scheduler, OS Resources and so on. In addition, one would use the SWC to describe the ports towards the RTE.

The I/O Hardware Abstraction is a part of BSW. It could be considered as a group of modules. Although IOHWAB is integration code, each module of IOHWAB could fit to the BSWDT. Today, it is known that this point is not sufficiently documented in the current specification.

However, it is agreed that ECU signal will be mapped to a VFB Port (See chapter 7.2 and chapter 7.4). Moreover, to describe the interfaces between an I/O Hardware Abstraction implementation and applicative Software Components implementations (above RTE), one shall use the Software Component Template.

The intention of this chapter is to summarize all recommendations to define Ports, Interfaces and all other Software Component like elements during configuration process.

7.8.2 Requirements

7.8.2.1 I/O Hardware Abstraction Ports definition

[SWS_IoHwAb_00075] ⌈The I/O Hardware Abstraction specification defines only recommendations for the Port usage. The instantiation of the Ports shall be done during the configuration process and is specific to the ECU electronic design.⌋

The I/O Hardware Abstraction proposes to create one Port for each ECU signal identified, exception made for ECU Diagnosis signals that are connected to ECU Output signals. A relationship between this ECU signal and the Port shall be created.

Example: The ECU has 10 Analog input pins, 15 PWM output pins, 15 Digital output pins. The I/O Hardware Abstraction defines at least one Port for each ECU signal. In this simple example, Ports are instantiated 40 times.

7.9 Examples

7.9.1 EXAMPLE 1: Use case of on-board hardware

This example is derived from a power supplier ECU.

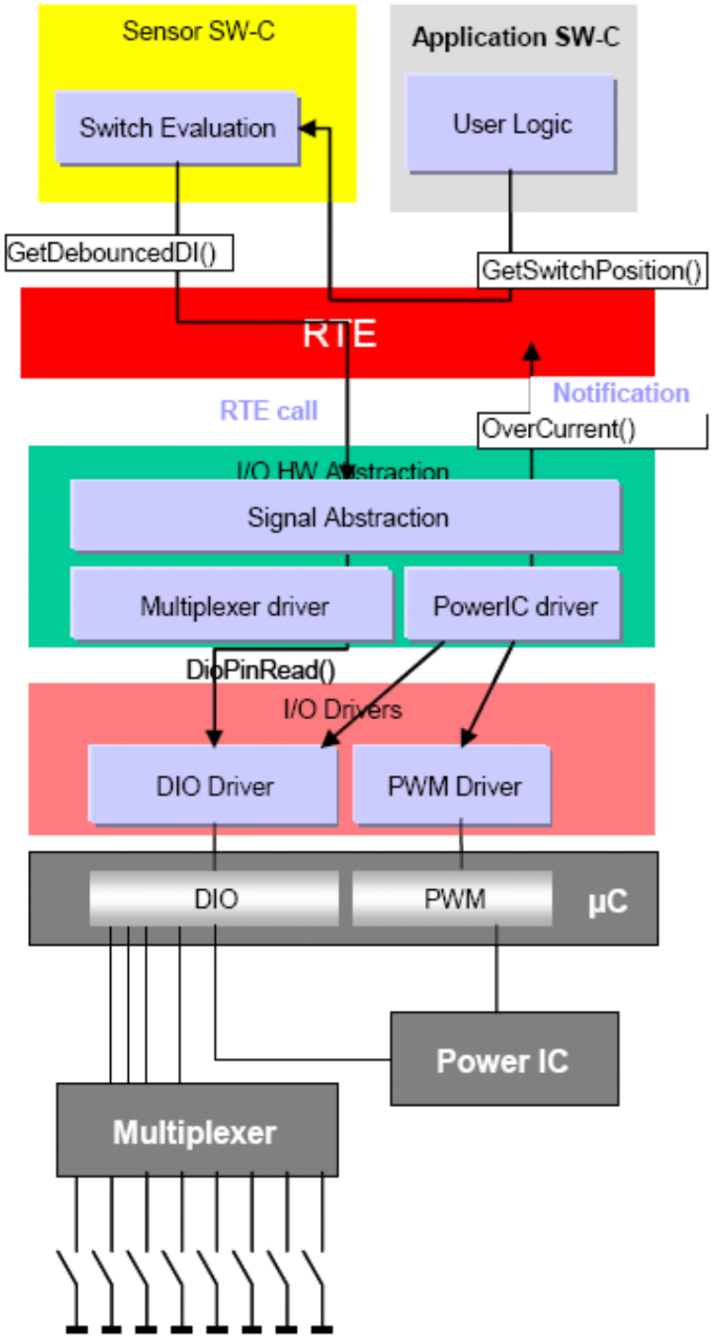

Figure 7.5: Use case of on-board hardware

The ECU has a high number of Digital Inputs (DI). One main group is the "**slow DI's**" for mechanical switches. The second main group is the "**fast DI's**" for the diagnosis of the Power IC (this DI indicates that the output current is to high "over current", these DI's are not led out of the ECU). The MCU has not enough PIN's -> the slow DI's are connected to 8 bit multiplexers (**3 address lines and 1 data line for each multiplexer**) the maximum time between the occurrence of an "over current" and the switch of the Power IC is 1 ms. One OEM requirement is that the reaction of a switch must be not later than **100 ms**. One other OEM requirement is that each DI must be debounced by **3 of 5 voting**. However the practice shows that the kind of debouncing is not really important because the mechanical switches and the power IC do not generate disturbing signals.

The **solution** today is that all DI (slow and fast) are read **every 0,8 ms (cyclic task)** (The scan rate for the slow DI could be lower but the overhead for an additional task is higher than the runtime savings). The debouncing for the slow DI's is **1 time in every loop** (so the worst cast delay to the debounced value is 3,2 ms). If an overcurrent is detected the pin will read again several times but in the same loop and the power IC will switched off immediately. The application runs **every 10 ms** and reads the debounced DI for the switches and the diagnosis information's.

Layer	*Multiplexed I/O*	*Power IC*
Application	Runnable reads the data every 10 ms	Gets a notification if the power IC detects overcurrent.
RTE	Handles runnables	
I/O Hardware Abstraction	8 signal mapped on ports, definition of port feature and Client/Server interface signal abstraction gives the debounce time (better than a debounce voting rule) A cyclic task performs a reading of input via DIO service call	I/O Hardware Abstraction makes decision to switch off the Power IC if an overcurrent is detected (in the driver of the external ASIC) A cyclic task performs a reading of input via DIO service call.
MCAL driver	DIO driver: address lines, 1 data line	DIO driver: 1 feedback line from power IC PWM driver: 1 line to the power IC
ECU hardware	Multiplexer: Mapping of 8 electrical signal	Power IC: Controls the power supply of the multiplexer

Table 7.1: Decomposition on the AUTOSAR architecture

7.9.2 EXAMPLE 2: Use case of failure monitoring

In this example, a diagnostic output signal shall be defined with the diagnosis attribute on the level of the I/O Hardware Abstraction.

Therefore, an input is used to perform the diagnosis of the output.

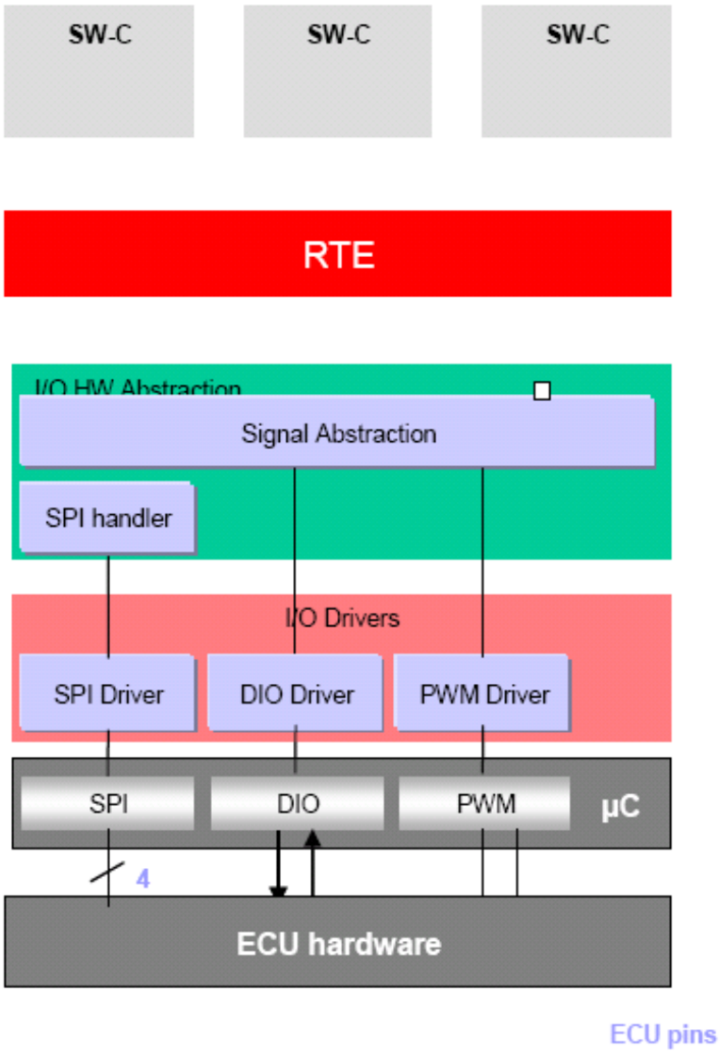

Figure 7.6: Use case of failure monitoring managed by SPI

When the I/O Hardware Abstraction asks for positioning one output (Dio_WriteChannel), a read-out of the channel is done via an ECU pin configured as input.

The ICU driver sends a notification to the I/O Hardware Abstraction.

The protection strategy is located in the integration code.

Software Component can get the diagnosis value through the port using the diagnosis operation.

7.9.3 EXAMPLE 3: Output power stage

The ECU hardware has a power stage ASIC.

Therefore, all ECU pins shall be available as "signals" at the level on the I/O Hardware Abstraction, just below the RTE.

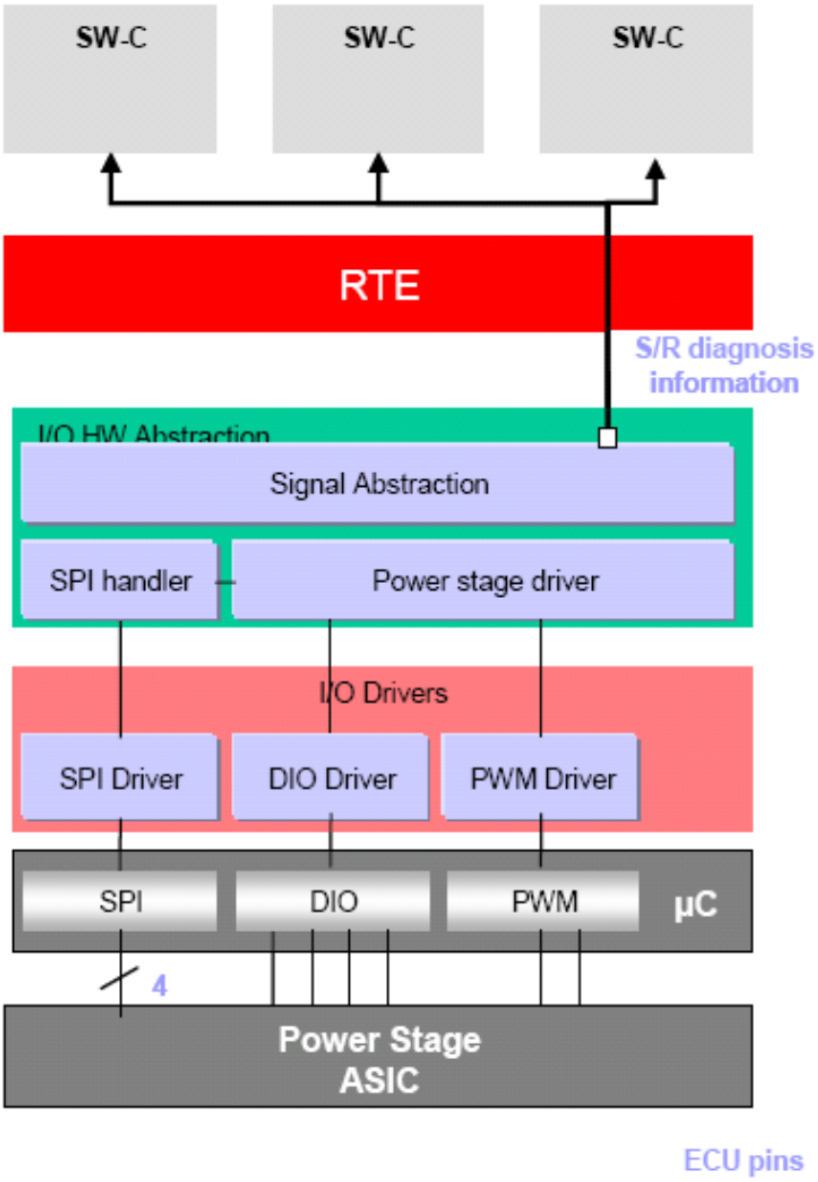

Figure 7.7: Use case of output power stage

Some outputs are controlled via the SPI driver/handler.

Some inputs are directly controlled via the DIO driver.

Some voltages, frequencies are set via the PWM driver.

A power stage driver provides the view of all outputs. It calls services of PWM, DIO drivers and SPI handler. The signal abstraction makes all these outputs "visible" from the point of view of Software Component (signals are mapped on Ports). The "Power stage driver" can be configurable.

Diagnosis: Every failure can be detected on the level of the power stage. The diagnosis data flow goes through the SPI communication to the Power stage driver. Then, the diagnosis is provided to all Software Component via an S/R interface.

7.9.4 EXAMPLE 4: Setting sensor and controlling periphery in low power state

The ECU controls a sensor through its ADC and its DIO Peripherals. Under specific circumstances, the ECU enters an operation mode in which the sensor is shut down and the ADC is set in low power state.

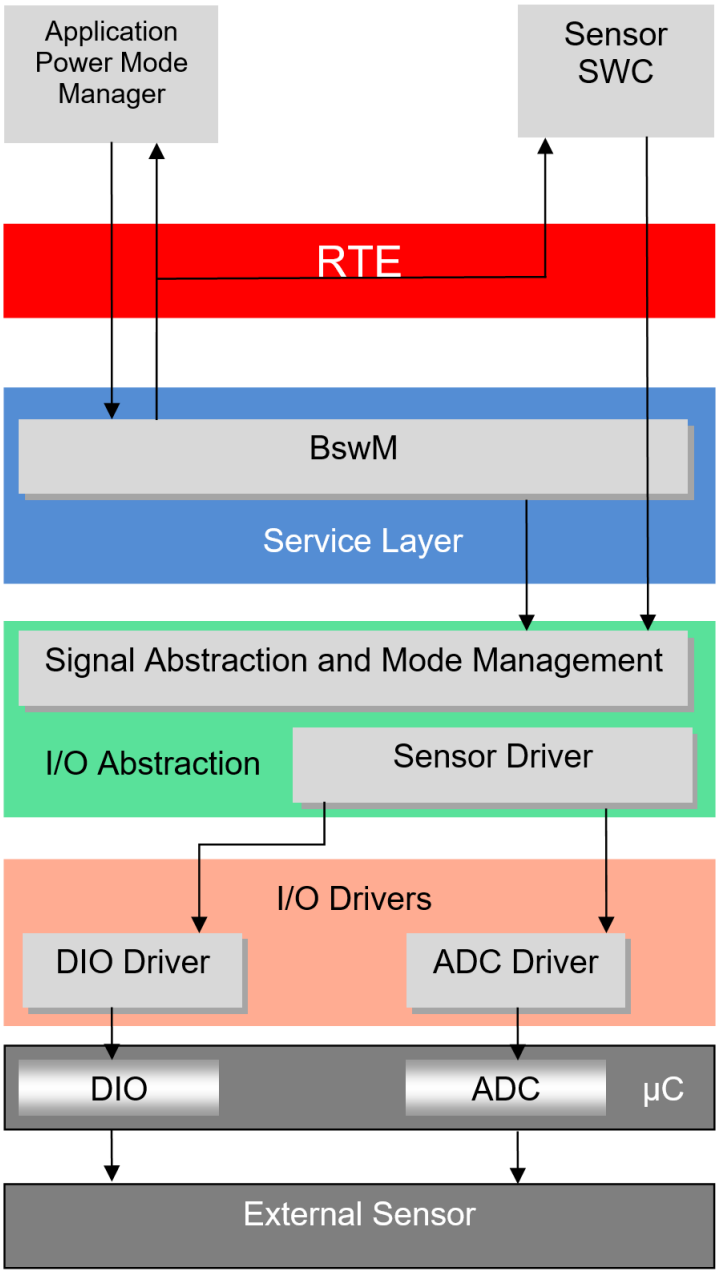

Figure 7.8: Use case low power mode setting

The sequence of actions is as follows:

The Application Power Mode Manager issues a Mode Request to BswM to switch to "LowPowerMode".

BswM evaluates the requests and, if the all pre-conditions are met, issues a mode switch to the Power Mode Manager and to the Sensor SWC.

The sensor SWC stops reading the sensory data (i.e. doesn't request any Get operation to the IoHwAbs anymore).

The IoHwAbs deregisters its notifications from the ADC and eventually stop HW cyclical acquisitions.

The IoHwAbs commands external sensory HW into a low power mode or shut it off.

The IoHwAbs calls its Low Power Mode preparation Callouts and then its Low Power Mode setting Callouts, as defined in the configuration in order to attain the ADC (in this case) power state related to the requested Application Low Power mode "LowPowerMode".

The process can be controlled step by step by introducing more fine granular mode requests and reacting on the acknowledgements and/or switches.

8 API specification

8.1 Imported types

In this chapter, all types included from the following modules are listed:

[SWS_IoHwAb_91005] Definition of imported datatypes of module IoHwAb ⌈

Module	*Header File*	*Imported Type*
Adc	Adc.h	Adc_GroupType
	Adc.h	Adc_StatusType
	Adc.h	Adc_StreamNumSampleType
	Adc.h	Adc_ValueGroupType
Dio	Dio.h	Dio_ChannelGroupType
	Dio.h	Dio_ChannelType
	Dio.h	Dio_LevelType
	Dio.h	Dio_PortLevelType
	Dio.h	Dio_PortType
EcuM	EcuM.h	EcuM_WakeupSourceType
Gpt	Gpt.h	Gpt_ChannelType
	Gpt.h	Gpt_ModeType
	Gpt.h	Gpt_ValueType
Icu	Icu.h	Icu_ActivationType
	Icu.h	Icu_ChannelType
	Icu.h	Icu_DutyCycleType
	Icu.h	Icu_EdgeNumberType
	Icu.h	Icu_IndexType
	Icu.h	Icu_InputStateType
	Icu.h	Icu_ValueType
Ocu	Ocu.h	Ocu_ChannelType
	Ocu.h	Ocu_PinStateType
	Ocu.h	Ocu_ReturnType
	Ocu.h	Ocu_ValueType
Port	Port.h	Port_PinDirectionType
	Port.h	Port_PinModeType
	Port.h	Port_PinType
Pwm	Pwm.h	Pwm_ChannelType
	Pwm.h	Pwm_OutputStateType
Spi	Spi.h	Spi_AsyncModeType
	Spi.h	Spi_ChannelType
	Spi.h	Spi_DataBufferType
	Spi.h	Spi_HWUnitType
	Spi.h	Spi_JobResultType
	Spi.h	Spi_JobType

▽

△

Module	*Header File*	*Imported Type*
	Spi.h	Spi_NumberOfDataType
	Spi.h	Spi_SeqResultType
	Spi.h	Spi_SequenceType
	Spi.h	Spi_StatusType
Std	Std_Types.h	Std_ReturnType
	Std_Types.h	Std_VersionInfoType

⌋

8.2 Type definitions

8.2.1 IoHwAb<Init_Id>_ConfigType

[SWS_IoHwAb_00157] Definition of datatype IoHwAb{Init_Id}_ConfigType

Upstream requirements: SRS_BSW_00414

⌈

Name	IoHwAb{Init_Id}_ConfigType	
Kind	Structure	
Elements	implementation specific	
	Type	–
	Comment	–
Description	Configuration data structure of the IoHwAb module.	
Available via	IoHwAb.h	

⌋

8.3 Function definitions

This is a list of functions provided for upper layer modules.

NOTE FOR I/O HARDWARE ABSTRACTION:

As explained in the previous chapters, no functional API will be specified for the I/O Hardware Abstraction.

8.3.1 IoHwAb_Init<Init_Id>

[SWS_IoHwAb_00119] Definition of API function IoHwAb_Init<Init_Id> ⌈

Service Name	IoHwAb_Init<Init_Id>	
Syntax	`void IoHwAb_Init<Init_Id> ( const IoHwAb{Init_Id}_ConfigType* ConfigPtr )`	
Service ID [hex]	0x01	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	ConfigPtr	Pointer to the selected configuration set.
Parameters (inout)	None	
Parameters (out)	None	
Return value	None	
Description	Initializes either all the IO Hardware Abstraction software or is a part of the IO Hardware Abstraction.	
Available via	IoHwAb.h	

⌋

[SWS_IoHwAb_00158]

Upstream requirements: SRS_BSW_00414

⌈The Configuration pointer ConfigPtr shall always have a `NULL_PTR` value.⌋

The Configuration pointer ConfigPtr is currently not used and shall therefore be set `NULL_PTR` value.

[SWS_IoHwAb_00059]

Upstream requirements: SRS_BSW_00101

⌈This kind of function initializes either all the I/O Hardware Abstraction software, or a part of the I/O Hardware Abstraction.⌋

[SWS_IoHwAb_00060]

Upstream requirements: SRS_BSW_00101

⌈The multiplicity of I/O devices managed by the I/O Hardware Abstraction software shall be handled via several init functions. Each init function shall be tagged with an <Init_ID>. Therefore, an external device, having its driver encapsulated inside the I/O Hardware Abstraction, can be separately initialized.⌋

[SWS_IoHwAb_00061]

Upstream requirements: SRS_BSW_00101

⌈This kind of init function shall called by the ECU State Manager. The ECU integrator is able to configure the init sequence order called by the ECU State manager.⌋

[SWS_IoHwAb_00102]

Upstream requirements: SRS_BSW_00441

⌈After having finished the module initialization, the I/O Hardware Abstraction state shall be set to `IOHWAB_IDLE`, the job result shall be set to `IOHWAB_JOB_OK`.⌋

8.3.2 IoHwAb_GetVersionInfo

[SWS_IoHwAb_00120] Definition of API function IoHwAb_GetVersionInfo ⌈

Service Name	IoHwAb_GetVersionInfo	
Syntax	`void IoHwAb_GetVersionInfo ( Std_VersionInfoType* versioninfo )`	
Service ID [hex]	0x10	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	None	
Parameters (inout)	None	
Parameters (out)	versioninfo	Pointer to where to store the version information of this implementation of IO Hardware Abstraction.
Return value	None	
Description	Returns the version information of this module.	
Available via	IoHwAb.h	

⌋

8.4 Callback notifications

This is a list of functions provided for lower layer modules.

8.4.1 IoHwAb_AdcNotification<#groupID>

[SWS_IoHwAb_00121] Definition of callback function IoHwAb_AdcNotification<#groupID> ⌈

Service Name	IoHwAb_AdcNotification<#groupID>
Syntax	`void IoHwAb_AdcNotification<#groupID> ( void )`
Service ID [hex]	0x20

▽

△

Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Will be called by the ADC Driver when a group conversion is completed for group <#groupID>.
Available via	IoHwAb_Adc.h

⌋

[SWS_IoHwAb_00104] ⌈The function IoHwAb_AdcNotification<#groupID> is intended to be called by the ADC driver when a group conversion is completed for group <#groupID>.⌋

8.4.2 IoHwAb_Pwm_Notification<#channel>

[SWS_IoHwAb_00122] Definition of callback function IoHwAb_PwmNotification<#channel> ⌈

Service Name	IoHwAb_PwmNotification<#channel>
Syntax	`void IoHwAb_PwmNotification<#channel> ( void )`
Service ID [hex]	0x30
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Will be called by the PWM Driver when a signal edge occurs on channel <#channel>.
Available via	IoHwAb_Pwm.h

⌋

[SWS_IoHwAb_00105] ⌈The function IoHwAb_PwmNotification<#channel> is intended to be called by the PWM driver when a signal edge occurs on channel <#channel>.⌋

8.4.3 IoHwAb_IcuNotification<#channel>

[SWS_IoHwAb_00123] Definition of callback function IoHwAb_IcuNotification<#channel> ⌈

Service Name	IoHwAb_IcuNotification<#channel>
Syntax	`void IoHwAb_IcuNotification<#channel> ( void )`
Service ID [hex]	0x40
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Will be called by the ICU driver when a signal edge occurs on channel <#channel>.
Available via	IoHwAb_Icu.h

⌋

[SWS_IoHwAb_00106] ⌈The function IoHwAb_IcuNotification<#channel> is intended to be called by the ICU driver when a signal edge occurs on channel <#channel>.⌋

8.4.4 IoHwAb_GptNotification<#channel>

[SWS_IoHwAb_00124] Definition of callback function IoHwAb_GptNotification<#channel> ⌈

Service Name	IoHwAb_GptNotification<#channel>
Syntax	`void IoHwAb_GptNotification<#channel> ( void )`
Service ID [hex]	0x50
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Will be called by the GPT driver when a timer value expires on channel <#channel>.
Available via	IoHwAb_Gpt.h

⌋

[SWS_IoHwAb_00107] ⌈The function IoHwAb_GptNotification<#channel> is intended to be called by the GPT driver when a timer value expires on channel <#channel>.⌋

8.4.5 IoHwAb_OcuNotification<#channel>

[SWS_IoHwAb_00155] Definition of callback function IoHwAb_OcuNotification<#channel> ⌈

Service Name	IoHwAb_OcuNotification<#channel>
Syntax	`void IoHwAb_OcuNotification<#channel> ( void )`
Service ID [hex]	0xa0
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Will be called by the OCU driver when the current value of the threshold matches the threshold on the channel<#channel>.
Available via	IoHwAb_Ocu.h

⌋

[SWS_IoHwAb_00156] ⌈The function IoHwAb_OcuNotification<#channel> is intended to be called by the OCU driver when the current value of the counter matches the threshold on channel <#channel>.⌋

8.4.6 IoHwAb_Pwm_NotifyReadyForPowerState<#MODE>

[SWS_IoHwAb_91002] Definition of API function IoHwAb_Pwm_NotifyReadyForPowerState<#Mode> ⌈

Service Name	IoHwAb_Pwm_NotifyReadyForPowerState<#Mode>
Syntax	`void IoHwAb_Pwm_NotifyReadyForPowerState<#Mode> ( void )`
Service ID [hex]	0x60
Sync/Async	Synchronous
Reentrancy	Non Reentrant

▽

△

Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	The API shall be invoked by the PWM Driver when the requested power state preparation for mode <#Mode> is completed.
Available via	IoHwAb_Pwm.h

⌋

This interface provided by CDD or IoHwAbs is needed if the PWM Driver is configured to support power state control in asynchronous mode.

8.4.7 IoHwAb_Adc_NotifyReadyForPowerState<#MODE>

[SWS_IoHwAb_00154] Definition of callback function IoHwAb_Adc_NotifyReady ForPowerState<#Mode> ⌈

Service Name	IoHwAb_Adc_NotifyReadyForPowerState<#Mode>
Syntax	`void IoHwAb_Adc_NotifyReadyForPowerState<#Mode> ( void )`
Service ID [hex]	0x70
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	The API shall be invoked by the ADC Driver when the requested power state preparation for mode <#Mode> is completed.
Available via	IoHwAb_Adc.h

⌋

This interface provided by CDD or IoHwAbs is needed if the ADC Driver is configured to support power state control in asynchronous mode.

8.5 Scheduled functions

These functions are directly called by Basic Software Scheduler. The following functions shall have no return value and no parameter. All functions shall be non-reentrant.

8.5.1 <Name of scheduled function>

Service name:	<Name of API call>
Service ID [hex]:	<Number of service ID. This ID is used as parameter for the error report API of Default Error Tracer. The ID shall not be equal to an ID within chapter [REF]>
Description:	<Set of local software requirements including ID that define the operation of this API call.>
Timing:	<fixed cyclic / variable cyclic / on pre condition>
Pre condition:	<List of assumptions about the environment in which the API call must operate.>
Configuration:	<Description of statically configurable attributes that affect this API call. For instance cycle time(s) in case of fixed cyclic timing.>

8.6 Functional Diagnostics Interface

This chapter describes the interface the I/O Hardware Abstraction provides to the DCM module to realize 'Functional Diagnostics of Software Components'.

'Functional Diagnostics of Software Components' means, that by the provided interface, the DCM module is able to control and read each implemented ECU signal.

8.6.1 IoHwAb_Dcm_<EcuSignalName>

[SWS_IoHwAb_00135] Definition of API function IoHwAb_Dcm_<EcuSignal Name>

Upstream requirements: SRS_IoHwAb_00002

⌈

Service Name	IoHwAb_Dcm_<EcuSignalName>	
Syntax	`void IoHwAb_Dcm_<EcuSignalName> ( uint8 action, <EcuSignalDataType> signal )`	
Service ID [hex]	0xB0	
Sync/Async	Synchronous	
Reentrancy	–	
Parameters (in)	action	IOHWAB_RETURNCONTROLTOECU: Unlock the signal IOHWAB_RESETTODEFAULT: Lock the signal and set it to a configured default value IOHWAB_FREEZECURRENTSTATE: Lock the signal to the current value IOHWAB_ SHORTTERMADJUSTMENT: Lock the signal and adjust it to a value given by the DCM module
	signal	Value to adjust the signal to (only used for 'short term adjustment').
Parameters (inout)	None	
Parameters (out)	None	

▽

△

Return value	None
Description	This function provides control access to a certain ECU Signal to the DCM module (<Ecu Signalname> is the symbolic name of an ECU Signal). The ECU signal can be locked and unlocked by this function. Locking 'freezes' the ECU signal to the current value, the configured default value or a value given by the parameter 'signal'.
Available via	IoHwAb_Dcm.h

⌋

[SWS_IoHwAb_00136]

Upstream requirements: SRS_IoHwAb_00002

⌈This function allows controlling the associated ECU Signal, i.e. the ECU Signal can be locked, unlocked, and adjusted to a certain value.⌋

[SWS_IoHwAb_00138]

Upstream requirements: SRS_IoHwAb_00002

⌈This function shall be pre compile time configurable On/Off.⌋

Locking a signal means, that the certain signal is software-locked towards the SW-C, i.e. the SW-C's requests have no effect on the hardware in the locked state. In case C/S-communication is used for input signals, it might be necessary to have an IoHwAb-internal buffer, whose value can be adjusted by the DCM.

8.6.2 IoHwAb_Dcm_Read<EcuSignalName>

[SWS_IoHwAb_00139] Definition of API function IoHwAb_Dcm_Read<EcuSignal Name>

Upstream requirements: SRS_IoHwAb_00002

⌈

Service Name	IoHwAb_Dcm_Read<EcuSignalName>	

Syntax	`void IoHwAb_Dcm_Read<EcuSignalName> (`
	`  <EcuSignalDataType>* signal`
	`)`	

Service ID [hex]	0xC0	
Sync/Async	Synchronous	
Reentrancy	–	
Parameters (in)	None	
Parameters (inout)	None	
Parameters (out)	signal	Pointer to the variable where the current signal value shall be stored
Return value	None	

▽

△

Description	This function provides read access to a certain ECU Signal to the DCM module (<Ecu Signalname> is the symbolic name of an ECU Signal).
Available via	IoHwAb_Dcm.h

⌋

[SWS_IoHwAb_00140]

Upstream requirements: SRS_IoHwAb_00002

⌈This function provides read access to a certain ECU Signal to the DCM module. The read access is independent from the ECU Signal's current state (locked/unlocked) and shall always read the current physical value from the hardware.⌋

[SWS_IoHwAb_00142]

Upstream requirements: SRS_IoHwAb_00002

⌈This function shall be pre compile time configurable On/Off.⌋

8.7 Power State Functions

8.7.1 IoHwAb_PreparePowerState<#MODE>

[SWS_IoHwAb_00146] Definition of API function IoHwAb_PreparePowerState<#Mode> ⌈

Service Name	IoHwAb_PreparePowerState<#Mode>

Syntax	`void IoHwAb_PreparePowerState<#Mode> (`
	`  void`
	`)`

Service ID [hex]	0x80
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	The API shall be invoked by the IoHwAbs in order to prepare the transition to a given power state. The aim of this API is to incapsulate all actions to prepare the HW for a predefined power mode, decoupling application power definition from HW power states.
Available via	IoHwAb.h

⌋

[SWS_IoHwAb_00149] ⌈This API is a configurable callout and shall be defined per configuration once per Power Mode to be managed.⌋

[SWS_IoHwAb_00150] ⌈This callout shall be executed in the context of the IoHwAbs SWC, meaning that it has full access to the MCAL.⌋

Many peripheral power state transition requests can be connected to a given Power Mode transition to be implemented by this callout, along with any other action needed to bring the peripherals in the desired power state (cross dependencies between peripherals can be solved in this context).

8.7.2 IoHwAb_ EnterPowerState <#MODE>

[SWS_IoHwAb_00147] Definition of API function IoHwAb_EnterPowerState<#Mode> ⌈

Service Name	IoHwAb_EnterPowerState<#Mode>

Syntax	`void IoHwAb_EnterPowerState<#Mode> (`
	`  void`
	`)`

Service ID [hex]	0x90
Sync/Async	Asynchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	The API shall be invoked by the IoHwAbs in order to effectively enter a power state which was prepared by the API IoHwAb_PreparePowerState<#Mode>() . The aim of this API is to incapsulate all actions to set the HW in a power state corresponding to a predefined power mode, decoupling application power definition from HW power states.
Available via	IoHwAb.h

⌋

[SWS_IoHwAb_00151] ⌈This API is a configurable callout and shall be defined per configuration once per Power Mode to be managed.⌋

[SWS_IoHwAb_00152] ⌈This callout shall be executed in the context of the IoHwAbs SWC, meaning that it has full access to the MCAL.⌋

[SWS_IoHwAb_00153] ⌈This API executes all power state transition prepared by the preceding call to the correposonding IoHwAb_PreparePowerState<#Mode>.⌋

8.8 Expected interfaces

In this chapter, all interfaces required from other modules are listed.

8.8.1 Mandatory Interfaces

There are no mandatory interfaces for I/O Hardware Abstraction. Which interfaces the I/O Hardware Abstraction uses depends on the expected functionality of the channels that are defined by the SWC.

Example of an I/O Hardware Abstraction using all MCAL drivers APIs :

Note that <module_name>_Init and <module_name>_DeInit functions are not listed below. The initialization sequence is called by the ECU state manager, and not by the I/O Hardware Abstraction.

<module_name>_GetVersionInfo functions are also not listed here.

This table has been built according to following documents

- Driver ADC document [10]
- Driver DIO document [11]
- Driver ICU document [12]
- Driver PWM document [13]
- Driver PORT document [14]
- Driver GPT document [15]
- Driver SPI document [16]
- Driver OCU document [17]

[SWS_IoHwAb_91003] Definition of mandatory interfaces required by module Io HwAb ⌈

API Function	*Header File*	*Description*
Adc_GetGroupStatus	Adc.h	Returns the conversion status of the requested ADC Channel group.
Adc_GetStreamLastPointer	Adc.h	Returns the number of valid samples per channel, stored in the result buffer. Reads a pointer, pointing to a position in the group result buffer. With the pointer position, the results of all group channels of the last completed conversion round can be accessed. With the pointer and the return value, all valid group conversion results can be accessed (the user has to take the layout of the result buffer into account).

▽

△

API Function	*Header File*	*Description*
Adc_ReadGroup	Adc.h	Reads the group conversion result of the last completed conversion round of the requested group and stores the channel values starting at the Data BufferPtr address. The group channel values are stored in ascending channel number order (in contrast to the storage layout of the result buffer if streaming access is configured).
Adc_SetupResultBuffer	Adc.h	Initializes ADC driver with the group specific result buffer start address where the conversion results will be stored. The application has to ensure that the application buffer, where DataBufferPtr points to, can hold all the conversion results of the specified group. The initialization with Adc_SetupResultBuffer is required after reset, before a group conversion can be started.
Adc_StartGroupConversion	Adc.h	Starts the conversion of all channels of the requested ADC Channel group.
Adc_StopGroupConversion	Adc.h	Stops the conversion of the requested ADC Channel group.
Dio_ReadChannel	Dio.h	Returns the value of the specified DIO channel.
Dio_ReadChannelGroup	Dio.h	This Service reads a subset of the adjoining bits of a port.
Dio_ReadPort	Dio.h	Returns the level of all channels of that port.
Dio_WriteChannel	Dio.h	Service to set a level of a channel.
Dio_WriteChannelGroup	Dio.h	Service to set a subset of the adjoining bits of a port to a specified level.
Dio_WritePort	Dio.h	Service to set a value of the port.
Gpt_CheckWakeup	Gpt.h	Checks if a wakeup capable GPT channel is the source for a wakeup event and calls the ECU state manager service EcuM_SetWakeupEvent in case of a valid GPT channel wakeup event.
Gpt_DisableWakeup	Gpt.h	Disables the wakeup interrupt of a channel (relevant in sleep mode).
Gpt_EnableWakeup	Gpt.h	Enables the wakeup interrupt of a channel (relevant in sleep mode).
Gpt_GetTimeElapsed	Gpt.h	Returns the time already elapsed.
Gpt_GetTimeRemaining	Gpt.h	Returns the time remaining until the target time is reached.
Gpt_SetMode	Gpt.h	Sets the operation mode of the GPT.
Icu_DisableEdgeCount	Icu.h	This function disables the counting of edges of the given channel.
Icu_DisableNotification	Icu.h	This function disables the notification of a channel.
Icu_DisableWakeup	Icu.h	This function disables the wakeup capability of a single ICU channel.
Icu_EnableEdgeCount	Icu.h	This function enables the counting of edges of the given channel.
Icu_EnableNotification	Icu.h	This function enables the notification on the given channel.
Icu_EnableWakeup	Icu.h	This function (re-)enables the wakeup capability of the given ICU channel.
Icu_GetDutyCycleValues	Icu.h	This function reads the coherent active time and period time for the given ICU Channel.
Icu_GetEdgeNumbers	Icu.h	This function reads the number of counted edges.
Icu_GetInputState	Icu.h	This function returns the status of the ICU input.

▽

△

API Function	*Header File*	*Description*
Icu_GetTimeElapsed	Icu.h	This function reads the elapsed Signal Low Time for the given channel.
Icu_GetTimestampIndex	Icu.h	This function reads the timestamp index of the given channel.
Icu_ResetEdgeCount	Icu.h	This function resets the value of the counted edges to zero.
Icu_SetActivationCondition	Icu.h	This function sets the activation-edge for the given channel.
Icu_StartSignalMeasurement	Icu.h	This function starts the measurement of signals.
Icu_StartTimestamp	Icu.h	This function starts the capturing of timer values on the edges.
Icu_StopSignalMeasurement	Icu.h	This function stops the measurement of signals of the given channel.
Icu_StopTimestamp	Icu.h	This function stops the timestamp measurement of the given channel.
Ocu_DisableNotification	Ocu.h	This service is used to disable notifications from an OCU channel.
Ocu_EnableNotification	Ocu.h	This service is used to enable notifications from an OCU channel.
Ocu_GetCounter	Ocu.h	Service to read the current value of the counter.
Ocu_SetAbsoluteThreshold	Ocu.h	Service to set the value of the channel threshold using an absolute input data.
Ocu_SetPinState	Ocu.h	Service to set immediately the level of the pin associated to an OCU channel.
Ocu_SetRelativeThreshold	Ocu.h	Service to set the value of the channel threshold relative to the current value of the counter.
Ocu_StartChannel	Ocu.h	Service to start an OCU channel.
Ocu_StopChannel	Ocu.h	Service to stop an OCU channel.
Port_RefreshPortDirection	Port.h	Refreshes port direction.
Port_SetPinDirection	Port.h	Sets the port pin direction
Port_SetPinMode	Port.h	Sets the port pin mode.
Pwm_GetOutputState	Pwm.h	Service to read the internal state of the PWM output signal.
Spi_AsyncTransmit	Spi.h	Service to transmit data on the SPI bus.
Spi_Cancel	Spi.h	Service cancels the specified on-going sequence transmission.
Spi_GetHWUnitStatus	Spi.h	This service returns the status of the specified SPI Hardware microcontroller peripheral.
Spi_GetJobResult	Spi.h	This service returns the last transmission result of the specified Job.
Spi_GetSequenceResult	Spi.h	This service returns the last transmission result of the specified Sequence.
Spi_GetStatus	Spi.h	Service returns the SPI Handler/Driver software module status.
Spi_MainFunction_Handling	SchM_Spi.h	–
Spi_ReadIB	Spi.h	Service for reading synchronously one or more data from an IB SPI Handler/Driver Channel specified by parameter.
Spi_SetAsyncMode	Spi.h	Service to set the asynchronous mechanism mode for SPI busses handled asynchronously.
Spi_SetupEB	Spi.h	Service to setup the buffers and the length of data for the EB SPI Handler/Driver Channel specified.

▽

△

API Function	*Header File*	*Description*
Spi_SyncTransmit	Spi.h	Service to transmit data on the SPI bus
Spi_WriteIB	Spi.h	Service for writing one or more data to an IB SPI Handler/Driver Channel specified by parameter.

⌋

8.8.2 Optional Interfaces

This chapter defines all interfaces, which are required to fulfill an optional functionality of the I/O Hardware Abstraction.

[SWS_IoHwAb_91004] Definition of optional interfaces requested by module Io HwAb ⌈

API Function	*Header File*	*Description*
Det_ReportError	Det.h	Service to report development errors.
EcuM_SetWakeupEvent	EcuM.h	Sets the wakeup event.

⌋

8.8.3 Job End Notification

None

9 Sequence diagrams

9.1 ECU-signal provided by the I/O Hardware Abstraction (example)

This sequence diagram explains the example of chapter 7.5.2.5.

In this example, the Sensor / Actuator Component is the client, the I/O Hardware Abstraction is the server.

The Sensor/Actuator Component asks for a new value of the af_pressure AUTOSAR signal that is an ECU signal on the level of the I/O Hardware Abstraction.

After Adc conversion is finished, a notification coming from MCAL driver is converted into a RTE event for the Sensor / Actuator Component. Then, it can perform a synchronous read of the value present in the af_pressure signal buffer.

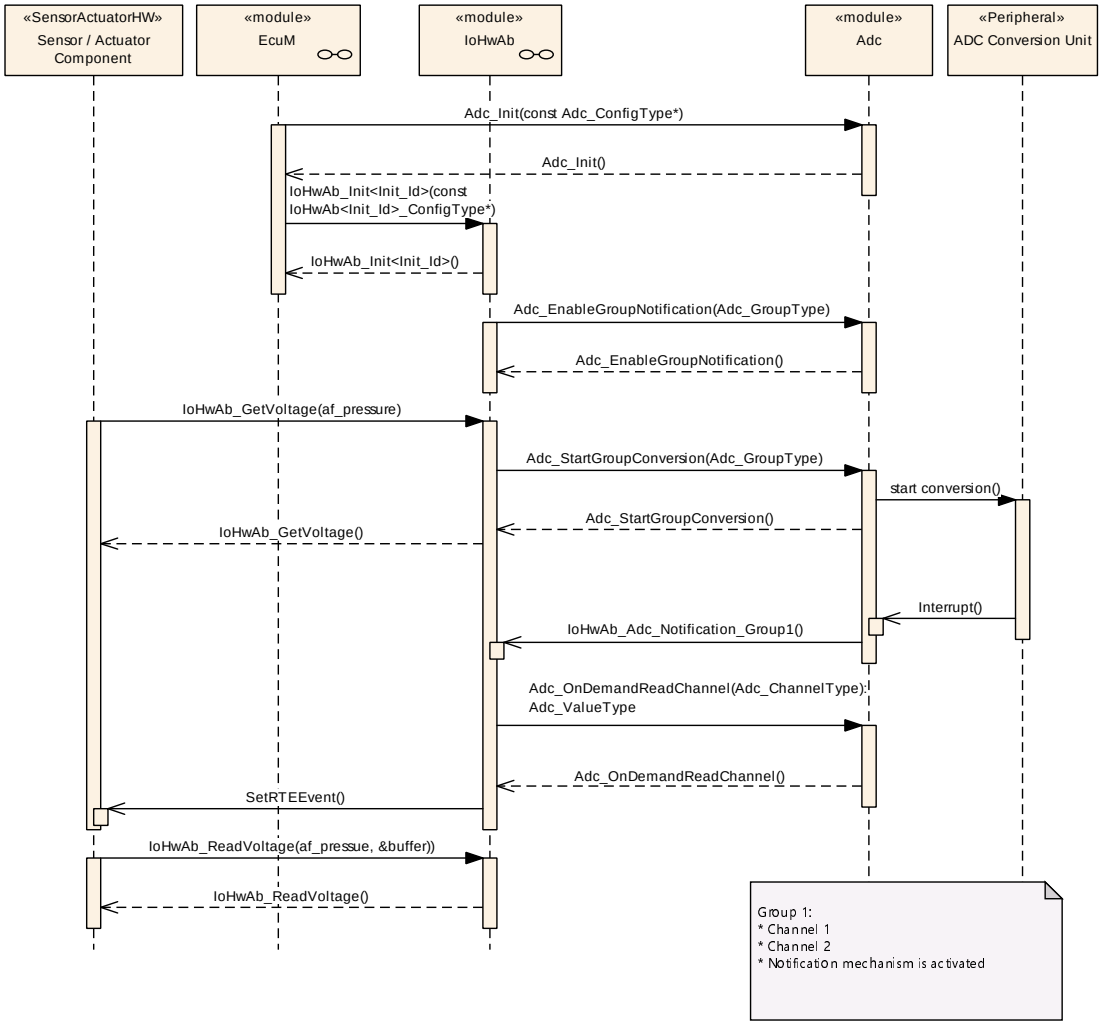

Figure 9.1: Sequence diagram - ADC conversion

Notes: APIs IoHwAb_GetVoltage(af_pressure) and IoHwAbReadVoltage(af_pressure, &buffer) are not specified interfaces, and are given only for the example.

The diagram in this example is intended to show the runnables and is not intended to show the server port to runnable mapping.

9.2 Setting ADC and PWM in a low consumption power state as a result of a request for an application low power mode (example)

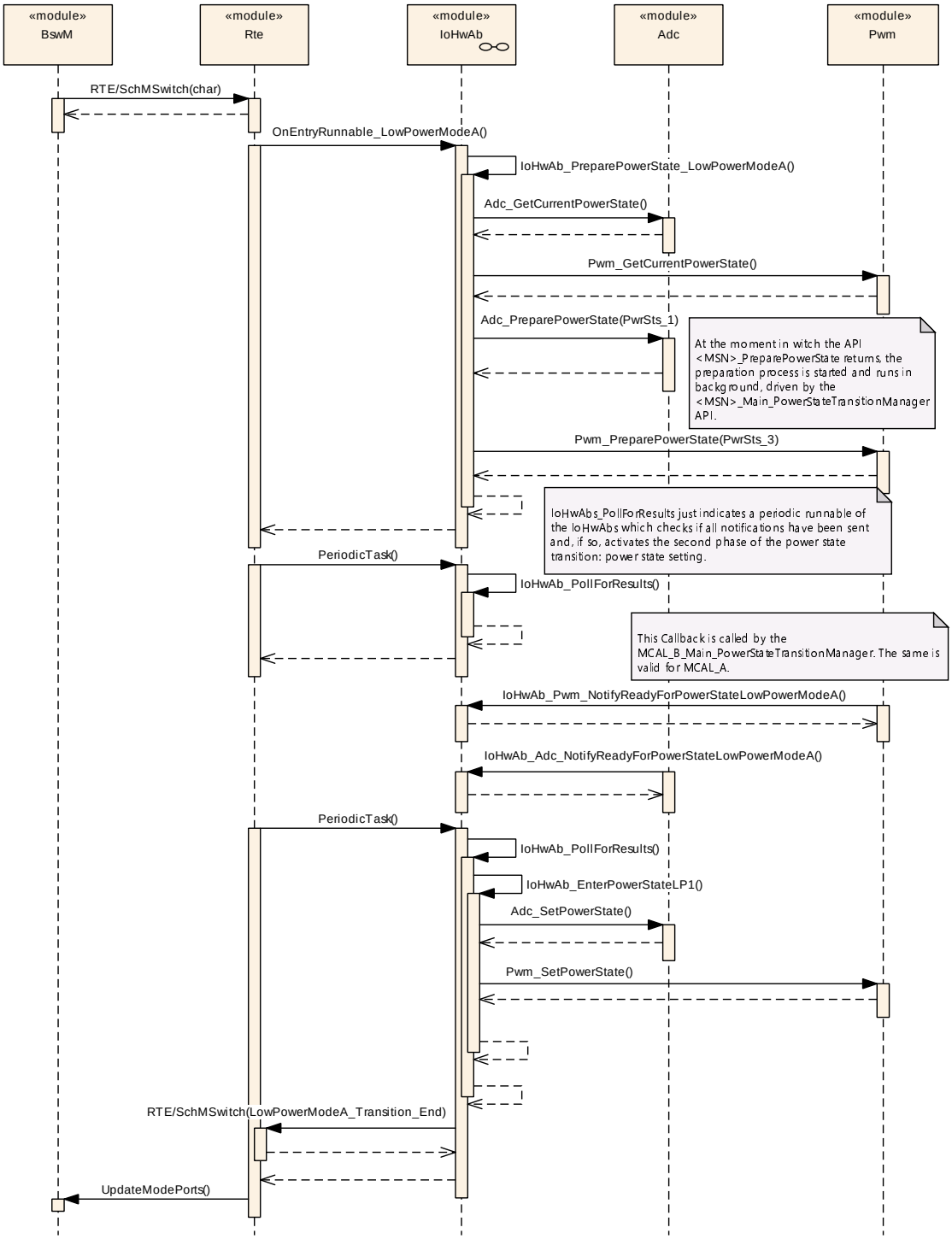

Figure 9.2: Asynchronous power state setting

The sequence diagram in figure 9.2 refers to a power state transition, where the peripherals are configured for asynchronous power state transitions. After having received

a request to prepare a power state, the peripheral's driver issues a notification to the caller (in this case IoHwAbs) to inform it of being ready to transition to the new power state.

In the following sequence diagram a synchronous transition is shown (the peripheral is immediately ready to transition, as soon as the preparation APIs return):

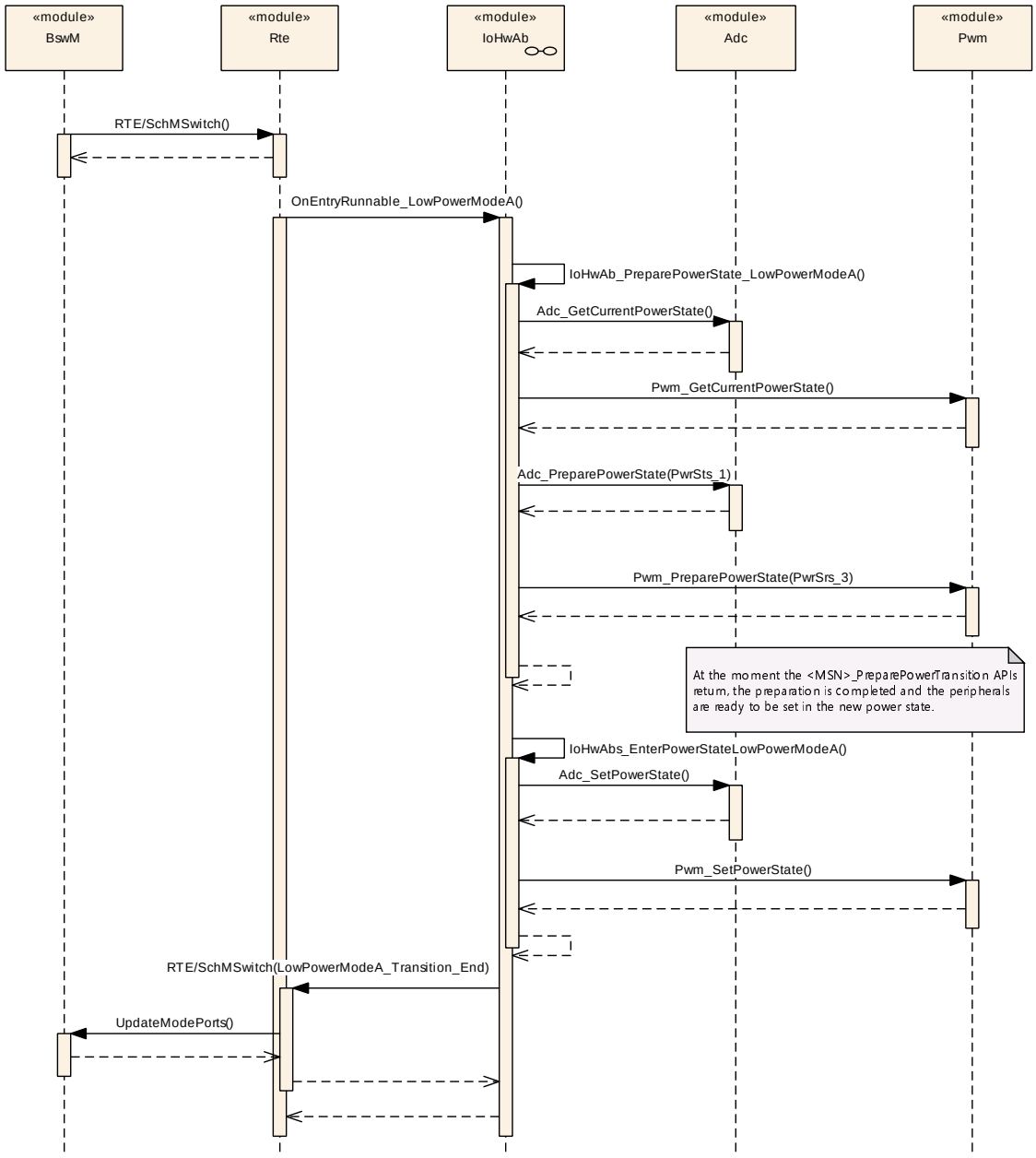

Figure 9.3: Synchronous power state setting

10 Configuration specification

The I/O Hardware Abstraction has no standardized configuration parameters and is therefore not part of the AUTOSAR ECU-C Parameter Definition. All parameters are vendor specific parameters.

10.1 Published Information

For details refer to the chapter 10.3 “Published Information” in [4].

A Not applicable requirements

[SWS_IoHwAb_NA_00999]

Upstream requirements: SRS_BSW_00300, SRS_BSW_00321, SRS_BSW_00325, SRS_BSW_00341, SRS_BSW_00342, SRS_BSW_00343, SRS_BSW_00398, SRS_BSW_00399, SRS_BSW_00400, SRS_BSW_00404, SRS_BSW_00405, SRS_BSW_00416, SRS_BSW_00417, SRS_BSW_00424, SRS_BSW_00428, SRS_BSW_00432, SRS_BSW_00439, SRS_BSW_00005, SRS_BSW_00007, SRS_BSW_00160, SRS_BSW_00161, SRS_BSW_00162, SRS_BSW_00164, SRS_BSW_00167, SRS_BSW_00168, SRS_BSW_00170, SRS_SPAL_12057, SRS_SPAL_12063, SRS_SPAL_12064, SRS_SPAL_12067, SRS_SPAL_12068, SRS_SPAL_12069, SRS_SPAL_12075, SRS_SPAL_12077, SRS_SPAL_12078, SRS_SPAL_12092, SRS_SPAL_12125, SRS_SPAL_12129, SRS_SPAL_12163, SRS_SPAL_12169, SRS_SPAL_12263, SRS_SPAL_12264, SRS_SPAL_12265, SRS_SPAL_12267, SRS_SPAL_12461, SRS_SPAL_12462, SRS_SPAL_12463, SRS_SPAL_00157

⌈These requirements are not applicable to this specification.⌋