Document Title	General Specification of Basic Software Modules
Document Category	Software specification (SWS)
Document UID	578
Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R26-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2026-11-26	R26-11	AUTOSAR Release Management	• Simplification of Multiplicity Configuration Class and Value Configuration Class • Support for DMA transfer (draft) • Support for MISRA C:2025 • Removal of version check and extended production errors
2025-11-27	R25-11	AUTOSAR Release Management	• Add information about Libraries • Move the "List of Basic Software Modules" to its own chapter and add corresponding requirements • Add support for Timing Protection extension for aperiodic servers • Minor corrections / clarifications / editorial changes
2024-11-27	R24-11	AUTOSAR Release Management	• Removal of Transient Faults • Added Appendix "List of Basic Software Modules" • Minor corrections / clarifications / editorial changes
2023-11-23	R23-11	AUTOSAR Release Management	• Update permissions for GetApplicationID • Add chapter "Debugging, Tracing, and Profiling Support of BasicSoftwareComponents" • Minor corrections / clarifications / editorial changes



△

2022-11-24	R22-11	AUTOSAR Release Management	• Removed compiler abstraction • Add Security events chapter • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation • Changed Document Status from Final to published
2021-11-25	R21-11	AUTOSAR Release Management	• Introduce C99 standard in AUTOSAR • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation • Changed Document Status from Final to published
2020-11-30	R20-11	AUTOSAR Release Management	• Incorporation of Intrusion Detection System Manager concept • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation • Changed Document Status from Final to published
2019-11-28	R19-11	AUTOSAR Release Management	• Include guard for header files • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation • Changed Document Status from Final to published
2018-10-31	4.4.0	AUTOSAR Release Management	• Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2017-12-08	4.3.1	AUTOSAR Release Management	• Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2016-11-30	4.3.0	AUTOSAR Release Management	• Meta Data handling • Changed to MISRA C 2012 Standard • Debugging support was removed • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation

▽

△

2015-07-31	4.2.2	AUTOSAR Release Management	• Debugging support marked as obsolete • Minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2014-10-31	4.2.1	AUTOSAR Release Management	• Update in error handling classification • Update in initialization function requirements • Updated due to SupportForPBLAndPB-SECUConfiguration concept • Minor corrections / clarifications / editorial changes; For details please refer to the BWCStatement
2014-03-31	4.1.3	AUTOSAR Release Management	• Update of include file structure and required header files requirement specification • Update of inter-module version check: removed REVISION/PATCH_VERSION from the required check • Formating and spelling corrections
2013-10-31	4.1.2	AUTOSAR Release Management	• Moved declarations of MainFunctions and BswModuleClientServerEntrys from the module header files to RTE/BswScheduler • Modified the Published Information definitions • Added the NULL pointer checking mechanism description • Removed the "Fixed cyclic", "Variable cyclic" and "On pre condition" from the Scheduled Functions description • Editorial changes
2013-03-15	4.1.1	AUTOSAR Release Management	• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction and functional overview 9
1.1 Traceability . . . 9
1.2 Document conventions . . . 9
2 Acronyms and abbreviations 11
3 Related documentation 12
3.1 Input documents & related standards and norms . . . 12
4 Constraints and assumptions 14
4.1 Limitations . . . 14
4.2 Applicability to car domains . . . 14
5 Dependencies to other modules 15
5.1 File structure . . . 15
5.1.1 Module implementation prefix . . . 15
5.1.2 Module implementation files . . . 16
5.1.3 Imported and exported information . . . 17
5.1.4 BSW Module Description . . . 18
5.1.5 Module documentation . . . 18
5.1.6 Code file structure . . . 20
5.1.6.1 Implementation source . . . 20
5.1.6.2 Link time configuration source . . . 22
5.1.6.3 Post-build time configuration source . . . 23
5.1.6.4 Interrupt frame implementation source . . . 24
5.1.7 Header file structure . . . 25
5.1.7.1 Implementation header . . . 25
5.1.7.2 Application Header File . . . 28
6 Requirements Tracing 29
7 Functional specification 35
7.1 General implementation specification . . . 35
7.1.1 Conformance to MISRA C and C standard . . . 35
7.1.2 Conformance to AUTOSAR Basic Software Requirements . . . 35
7.1.3 Conformance to AUTOSAR Methodology . . . 36
7.1.4 Platform independency . . . 36
7.1.5 Configurability . . . 37
7.1.6 Various naming conventions . . . 38
7.1.7 Configuration parameters . . . 39
7.1.8 Shared code . . . 40
7.1.9 Global data . . . 40
7.1.10 Usage of macros and inline functions . . . 41
7.1.11 Calling Scheduled functions (Main processing functions) . . . 41

7.1.12 Exclusive areas . . . 42
7.1.13 Callouts . . . 42
7.1.14 AUTOSAR Interfaces . . . 42
7.1.15 Interrupt service routines . . . 43
7.1.16 Restricted OS functionality access . . . 44
7.1.17 Access to hardware registers . . . 47
7.1.18 Data types . . . 48
7.1.18.1 AUTOSAR Standard Types . . . 48
7.1.18.2 Platform Specific Types . . . 48
7.1.19 Distributed execution on multi-partitioned systems . . . 50
7.1.20 Libraries . . . 51
7.1.20.1 Background . . . 51
7.1.20.2 Configuration . . . 51
7.1.20.3 Requirements . . . 52
7.1.21 Asynchronous copy approach (DRAFT) . . . 53
7.1.21.1 Background . . . 53
7.1.21.2 Requirements . . . 57
7.2 Error Handling . . . 58
7.2.1 Classification . . . 58
7.2.2 Development errors . . . 58
7.2.2.1 Synopsis . . . 58
7.2.2.2 Documentation . . . 59
7.2.2.3 Configuration . . . 59
7.2.2.4 Reporting . . . 60
7.2.3 Runtime errors . . . 61
7.2.3.1 Synopsis . . . 61
7.2.3.2 Documentation . . . 62
7.2.3.3 Configuration . . . 62
7.2.3.4 Reporting . . . 62
7.2.4 Production errors . . . 62
7.2.4.1 Synopsis . . . 62
7.2.4.2 Documentation . . . 63
7.2.4.3 Configuration . . . 63
7.2.4.4 Reporting . . . 64
7.2.4.5 Example use case: Error is detected and notified . . . 66
7.2.5 Specific topics . . . 66
7.2.5.1 Implementation specific errors . . . 66
7.2.5.2 Handling of Symbolic Name Values . . . 66
7.2.5.3 Handling of corrupted NV data . . . 68
7.3 Security events . . . 69
7.3.1 Synopsis . . . 69
7.3.2 Documentation . . . 69
7.3.3 Configuration . . . 69

7.3.4 Reporting . . . 70
7.4 Meta Data Handling . . . 70
8 API specification 73
8.1 Imported types . . . 73
8.2 Type definitions . . . 73
8.3 Function definitions . . . 75
8.3.1 General specification on API functions . . . 75
8.3.2 Initialization function . . . 77
8.3.3 De-Initialization function . . . 78
8.3.4 Callback notifications . . . 80
8.4 Scheduled functions . . . 80
8.5 Expected Interfaces . . . 82
8.5.1 Mandatory Interfaces . . . 82
8.5.2 Optional Interfaces . . . 82
8.5.3 Callout interfaces . . . 82
8.5.4 Configurable interfaces . . . 83
8.6 Service Interfaces . . . 83
9 Sequence diagrams 84
10 Configuration specification 85
10.1 Introduction to configuration specification . . . 85
10.1.1 Configuration and configuration parameters . . . 85
10.1.2 Variants . . . 85
10.1.3 Containers . . . 86
10.1.4 Configuration parameter tables . . . 86
10.1.4.1 General section: . . . 87
10.1.4.2 Configuration parameter section: . . . 87
10.1.4.3 Section of included/referenced containers: . . . 88
10.2 General configuration specification . . . 88
10.2.1 Configuration files . . . 88
10.2.2 Implementation names for configuration parameters . . . 88
10.2.3 Pre-compile time configuration . . . 89
10.2.4 Link time configuration . . . 89
10.2.5 Post-build time configuration . . . 90
10.2.6 Configuration variants . . . 91
10.3 Published Information . . . 92
11 List of Basic Software Modules 95
11.1 Libraries . . . 95
11.2 Modules . . . 95
11.3 Special Files . . . 98

A Change History 99
A.1 Change History of this document according to AUTOSAR Release R22-11 99
A.1.1 Added Specification Items in R22-11 99
A.1.2 Changed Specification Items in R22-11 99
A.1.3 Deleted Specification Items in R22-11 99
A.2 Change History of this document according to AUTOSAR Release R23-11 99
A.2.1 Added Specification Items in R23-11 99
A.2.2 Changed Specification Items in R23-11 99
A.2.3 Deleted Specification Items in R23-11 99
A.3 Change History of this document according to AUTOSAR Release R24-11 100
A.3.1 Added Specification Items in R24-11 100
A.3.2 Changed Specification Items in R24-11 100
A.3.3 Deleted Specification Items in R24-11 100
A.4 Change History of this document according to AUTOSAR Release R25-11 100
A.4.1 Added Specification Items in R25-11 100
A.4.2 Changed Specification Items in R25-11 100
A.4.3 Deleted Specification Items in R25-11 100
A.5 Change History of this document according to AUTOSAR Release R26-11 100
A.5.1 Added Specification Items in R26-11 100
A.5.2 Changed Specification Items in R26-11 101
A.5.3 Deleted Specification Items in R26-11 101

1 Introduction and functional overview

This document is the general basic software specification on AUTOSAR Basic Software modules. It complements the specification of BSW modules with as a common specification, which is valid for various BSW modules.

1.1 Traceability

The *Specification items* from this document describe the work products from the *BSW Module* implementation or their parts with regard to the *Basic Software Requirements*, which are described in AUTOSAR General Requirements on Basic Software Modules [1].

For every *BSW Module*, the traceability between *Specification items* and *Basic Software Requirements* is in scope of this document and the according *BSW Module* Software Specification. See also chapter 6 - ***Requirements traceability***.

The *BSW Module* implementation must guarantee traceability to the corresponding *Specification items* of this document and of the corresponding *BSW Module* specification.

Some *Specification items* are not applicable to every *BSW Module*. In such a case, its description explicitly mentions the condition for its applicability. If no condition is mentioned, the *Specification item* is applicable for all *BSW Modules*.

Please refer to AUTOSAR Standardization Template [2] Chapter 3.2.8 *"Meta-classes supporting Traceable Items"* for further information.

1.2 Document conventions

Code examples, symbols and other technical terms in general are typeset in monospace font, e.g. `const`.

Terms and expressions defined in AUTOSAR Glossary [3], within this specification (see chapter 2 - ***Acronyms and abbreviations***) or in related documentation are typeset in italic font, e.g. *Module implementation prefix*.

The *Basic Software Requirements* are described in document SRS BSW General [1]. These are referenced using SRS_BSW_<n> where <n> is its requirement id. For instance: SRS_BSW_00009.

Every *Specification item* starts with **[SWS_BSW_<nr>]**, where <nr> is its unique identifier number of the Specification item. This number is followed by the *Specification item* title. The scope of the *Specification item* description is marked with half brackets and is followed by the list of related requirements from SRS BSW General, between braces.

Example:

[SWS_BSW_<nr>] Specification item title

⌈Specification item description.⌋(SRS_BSW_<nr1>, SRS_BSW_<nr2>)

References to *Specification items* from other AUTOSAR documents use the conventions from the according document, for instance [SWS_CANIF_00001].

2 Acronyms and abbreviations

The glossary below includes acronyms and abbreviations relevant to the AUTOSAR Operating System module that are not included in the [3, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
BSW driver	For a list of BSW drivers see Chapter 11 “List of Basic Software Modules”, column "AUTOSAR SW layer".
Camel case	This document does not aim to specify rules for the camel case notation. Definition of CamelCase according to Wikipedia: "Camel case (...) *is the practice of writing phrases without spaces or punctuation, indicating the separation of words with a single capitalized letter, and the first word starting with either case. (...).*" Example: MainFunction
`<Ie>`	Implementation specific file name extension, see [SWS_BSW_00103].
`<Ma>`	*Module abbreviation*, see [SWS_BSW_00101].
`<MA>`	*Capitalized module abbreviation*. The *Capitalized module abbreviation* `<MA>` is the *Module abbreviation* `<Ma>` completely written in upper case.
MCAL	The MCAL, Microcontroller Abstraction Layer, is defined in AUTOSAR Layered Software Architecture [4]
`<Mip>`	*Module implementation prefix*, see [SWS_BSW_00102].
`<MIP>`	*Capitalized module implementation prefix*. The *Capitalized module implementation prefix* `<MIP>` is the *Module implementation prefix* `<Mip>` ([SWS_BSW_00102]) completely written in upper case.
Module implementation prefix	*Module implementation prefix*, see [SWS_BSW_00102].
Module abbreviation	*Module abbreviation*, see [SWS_BSW_00101].
SWCD	Software Component Description

Table 2.1: Acronyms and abbreviations

3 Related documentation

3.1 Input documents & related standards and norms

[1] General Requirements on Basic Software Modules
AUTOSAR_CP_RS_BSWGeneral

[2] Standardization Template
AUTOSAR_FO_TPS_StandardizationTemplate

[3] Glossary
AUTOSAR_FO_TR_Glossary

[4] Layered Software Architecture
AUTOSAR_CP_EXP_LayeredSoftwareArchitecture

[5] Basic Software Module Description Template
AUTOSAR_CP_TPS_BSWModuleDescriptionTemplate

[6] Specification of ECU Configuration
AUTOSAR_CP_TPS_ECUConfiguration

[7] Methodology for Classic Platform
AUTOSAR_CP_TR_Methodology

[8] Specification of Memory Mapping
AUTOSAR_CP_SWS_MemoryMapping

[9] Specification of RTE Software
AUTOSAR_CP_SWS_RTE

[10] MISRA C:2025 Guidelines for the use of the C language in critical systems, ISBN-13: 978-1911700197

[11] Software Component Template
AUTOSAR_CP_TPS_SoftwareComponentTemplate

[12] Specification of Operating System
AUTOSAR_CP_SWS_OS

[13] Specification of Platform Types for Classic Platform
AUTOSAR_CP_SWS_PlatformTypes

[14] Specification of Default Error Tracer
AUTOSAR_CP_SWS_DefaultErrorTracer

[15] Specification of Diagnostic Event Manager
AUTOSAR_CP_SWS_DiagnosticEventManager

[16] Specification of NVRAM Manager
AUTOSAR_CP_SWS_NVRAMManager

[17] NV Data Handling Guideline

AUTOSAR_CP_EXP_NVDataHandling

4 Constraints and assumptions

4.1 Limitations

This specification is common to all AUTOSAR *BSW Modules* (see Chapter 11 "List of Basic Software Modules") and contains only general *Specification items* on *BSW Modules*. Some of these specification items may not be relevant to particular *BSW Modules*, whenever the conditions specified are not fulfilled.

4.2 Applicability to car domains

This document can be used for all domain applications when AUTOSAR Basic Software modules are used.

5 Dependencies to other modules

This specification is common to all AUTOSAR *BSW Modules* (see Chapter 11 “List of Basic Software Modules”) and contains only general *Specification items*, which complement every single *BSW Module* specification. It shall not be used as a standalone specification.

Example: The CAN Interface module is specified by this specification (*General Specification for BSW Modules*) and by the document *Specification on CAN Interface* (SWS CAN Interface).

5.1 File structure

This specification does not completely define the *BSW Module* file structure. Nevertheless, names of implementation files not specified here must anyway follow [SWS_BSW_00103].

5.1.1 Module implementation prefix

The *BSW Module implementation prefix* is used to form various identifiers used in work products of the *BSW Module* implementation, e.g. API names, parameter names, symbols and file names. This prefix is mainly formed by the *Module abbreviation* and, when necessary, additional vendor specific information.

The list of *Module abbreviations* is available in Chapter 11 “List of Basic Software Modules” within the column "Module Abbreviation".

[SWS_BSW_00101] Module abbreviation

Upstream requirements: SRS_BSW_00300

⌈The *Module abbreviation* <Ma> of a *BSW Module* shall be the same as defined in Appendix "List of Basic Software Modules".⌋

The *Capitalized module abbreviation* <MA> is the *Module abbreviation* completely written in upper case.

Examples of BSW *Module abbreviations*: EcuM, CanIf, OS, Com. The corresponding *Capitalized module abbreviations* are ECUM, CANIF, OS, COM.

[SWS_BSW_00102] Module implementation prefix

Upstream requirements: SRS_BSW_00300, SRS_BSW_00347

⌈*The Module implementation prefix* <Mip> shall be formed in the following way:

<Ma>[_<vi>_<ai>]

Where <Ma> is the *Module abbreviation* of the *BSW Module* ([SWS_BSW_00101]), <vi> is its vendorId and <ai> is its vendorApiInfix. The sub part in square brackets [_<vi>_<ai>] is omitted if no vendorApiInfix is defined for the *BSW Module*. For Complex Drivers and transformers, the <Mip> is directly derived from the apiServicePrefix.⌋

The elements vendorId and vendorApiInfix are defined in *BSW Module Description Template* [5]. Their usage may be obligatory in some situations, like in case of multiple instantiation of BSW Driver modules. These constraints are not in scope of SWS BSW General.

The element apiServicePrefix is defined in *Specification of ECU Configuration* [6].

The *Capitalized module implementation prefix* <MIP> is the *Module implementation prefix* completely written in upper case.

In some situations, the *Module implementation prefix* is written in the same way as the *Module abbreviation*. Nevertheless, their meanings are different: The usage of *Module implementation prefix* is requested whenever a differentiation within the same module type could be necessary, e.g. to differentiate symbols from different module instances.

Examples of *Module implementation prefixes*:

- FrIf: Prefix for FlexRay Interface module implementation, where no vendorId and vendorApiInfix are defined.
- Eep_21_LDExt: Prefix for EEPROM driver implementation, where vendorApiInfix and vendorId are identified by "LDExt" and "21" respectively.

Examples of *Module abbreviations*:

- FrIf: FlexRay Interface module abbreviation
- Eep: EEPROM driver module abbreviation

5.1.2 Module implementation files

This specification defines the following file types. Some of these types are mandatory for all *BSW Modules*, other depend on the according *BSW Module* specification:

File type, for all *BSW Modules*	Classification	Example: Com
Module documentation	mandatory	Not defined.
BSW Module description	mandatory	Not defined. See [5].
Implementation source	mandatory	Com.c
Implementation header	mandatory	Com.h
Pre-compile time configuration source	conditional	Com_Cfg.c
Link time configuration source	conditional	Com_Lcfg.c
Post-build time configuration source	conditional	Com_PBcfg.c
Interrupt frame implementation source	conditional	Gpt_Irq.c

Table 5.1: Module Implementation Files

Note that according to AUTOSAR Methodology [7] it is possible to deliver a *BSW Module* with its object files and only part of the source code. See also [SWS_BSW_00117].

[SWS_BSW_00103] General file naming convention

Upstream requirements: SRS_BSW_00300

⌈The name of all *BSW Module* implementation files shall be formed in the following way:

`<Mip>[_<Ie>]*.*`

The sup-part in square brackets `[_<Ie>]` is an optional implementation specific file name extension. The first wildcard * is empty or replaced according to the different types of files specified for the module, e.g. _Cfg, _Lcfg, _PBcfg, _Irq. The second wildcard * is replaced by the file name extension.⌋

Example:

Implementation sources for Can Interface module with vendor specific file name extensions added: `CanIf_MainFncs.c`, `CanIf_Api.c`.

[SWS_BSW_00170] File names are case sensitive

Upstream requirements: SRS_BSW_00464

⌈File names shall be considered case sensitive regardless of the file system in which they are used.⌋

[SWS_BSW_00171] File names are non-ambiguous

Upstream requirements: SRS_BSW_00465

⌈It shall not be allowed to name any two files so that they only differ by the case of their letters.⌋

5.1.3 Imported and exported information

[SWS_BSW_00104] Restrict imported information

Upstream requirements: SRS_BSW_00301

⌈The *BSW Module* shall import only the necessary information (i.e. header files) that is required to fulfill its functional requirements.⌋

Note that the availability of other modules in the basic software depends on the used configuration. This has to be considered before including header files of these modules.

Example: The BSW module implementation is generated by an AUTOSAR toolchain. The module generator has to check before including header files of other modules if the respective module is available in the system according to the used configuration.

[SWS_BSW_00105] Restrict exported information

Upstream requirements: SRS_BSW_00302

⌈The *BSW Module* shall export only that kind of information in their corresponding header files that is explicitly needed by other modules.⌋

This is necessary to avoid modules importing or exporting functionality that could be misused. Also compile time might possibly be shortened through this restriction.

Example: The *NVRAM Manager* does not need to know all processor registers just because some implementation has included the processor register file in another header file used by the *NVRAM Manager*.

Note:
After the module configuration, some imported or exported information may also become unnecessary, as part of the implementation may be disabled.

5.1.4 BSW Module Description

[SWS_BSW_00001] Provide BSW Module description

Upstream requirements: SRS_BSW_00423, SRS_BSW_00426, SRS_BSW_00427, SRS_BSW_00334

⌈The *BSW Module description* (`.arxml`) shall be provided for the module according to the AUTOSAR Specification of BSW Module Description Template [5].⌋

This specification does not define any file of the package structure for the *BSW Module Description*, as this delivery is specified in *AUTOSAR Specification of BSW Module Description Template* [5].

5.1.5 Module documentation

[SWS_BSW_00002] Provide BSW Module documentation

Upstream requirements: SRS_BSW_00009, SRS_BSW_00010

⌈The *BSW Module documentation* shall be provided with the *BSW Module* implementation.

The following content shall be part of it:

- Cover sheet with title, version number, date, company, document status, document name;
- Change history with version number, date, company, change description, document status;
- Table of contents (navigable);

- Functional overview;
- Source file list and description;
- Deviations to specification
- Deviations to requirements;
- Used resources (interrupts, μC peripherals etc.);
- Integration description (OS, interface to other modules etc.);
- Configuration description with parameter, description, unit, valid range, default value, relation to other parameters.
- Examples for:
  - The correct usage of the API;
  - The configuration of the module.

The following content may be part of it:

- Memory footprint (RAM, ROM, stack size) together with the module configuration, platform information, compiler and compiler options, which were used for the calculation.

⌋

If possible the Memory footprint documentation may include a dependency formula between configuration elements and used memory (e.g. each configured DTC additionally requires x bytes ROM and y bytes RAM).

[SWS_BSW_00003] Provide information on supported microcontroller and used tool chain

Upstream requirements: SRS_BSW_00341

⌈If the *BSW Module* implementation depends on microcontroller, then the *BSW Module documentation* shall also contain the following information:

- Microcontroller vendor
- Microcontroller family
- Microcontroller derivative
- Microcontroller stepping (mask revision), if relevant
- Tool chain name and version
- Tool chain options which were used for development / qualification of module

⌋

The scheduling strategy that is built inside the *BSW Modules* shall be compatible with the strategy used in the system. To achieve this, the scheduling strategy of module implementation shall be accordingly documented:

[SWS_BSW_00054] Document calling sequence of *Scheduled functions*

Upstream requirements: SRS_BSW_00428

⌈The *BSW Module documentation* shall provide information about the execution order of his *Scheduled functions*, i.e. for every one of these functions, if it has to be executed in a specific order or sequence with respect to other *BSW Scheduled* function (or functions).⌋

The *BSW Module* own specification provides further details on the intended sequence order of its *Scheduled functions*. This information shall be considered in documentation either.

[SWS_BSW_00061] Document configuration rules and constraints

Upstream requirements: SRS_BSW_00167

⌈The *BSW Module* implementation shall provide configuration rules and constraints in the *Module documentation* to enable plausibility checks of configuration during ECU configuration time where possible.⌋

5.1.6 Code file structure

The code file structure for the *BSW Module implementation* is provided in this chapter. Note that the file structure delivered to user may be different.

Example:

Source code is not delivered; various post-build configuration sets are delivered.

5.1.6.1 Implementation source

The Implementation source provides the implementation for functionality of the *BSW Module*.

[SWS_BSW_00004] Provide *Implementation source* files

Upstream requirements: SRS_BSW_00346

⌈The code file structure shall contain one or more files for the implementation of the provided *BSW Module* functionality: the *Implementation source* files. The file names shall be formed in the following way:

`<Mip>[_<Ie>].c`⌋

[SWS_BSW_00060] Declarations within *Implementation source* files are restricted

Upstream requirements: SRS_BSW_00302

⌈The *Implementation source* files of the *BSW Module* shall declare all constants, global data types and functions that are only used by the module internally. Pre-link time configuration parameters are an exception of this rule.⌋

To allow the compiler to check for consistency between declaration and definition of global variables and functions, the *Implementation source* shall include its own header file.

[SWS_BSW_00005] Include *Implementation header*

Upstream requirements: SRS_BSW_00346

⌈The module *Implementation source* files of the *BSW Module* shall include its own *Implementation header*.⌋

The *Memory mapping header* is necessary to enable the *BSW Module* to access the module specific functionality provided by the *BSW Memory Mapping* [8].

[SWS_BSW_00006] Include *Memory mapping header*

Upstream requirements: SRS_BSW_00437

⌈The *Implementation source* files of the *BSW Module* shall include the *BSW Memory mapping header* (`<Mip>_MemMap.h`).⌋

The *Module interlink header* is necessary in order to access the module specific functionality provided by the BSW Scheduler.

[SWS_BSW_00007] Include *Module interlink header*

Upstream requirements: SRS_BSW_00415

⌈If the *BSW Module* uses BSW Scheduler API or if it implements `BswSchedulableEntitys`, then the corresponding *Implementation source* files shall include the *Module interlink header* file in order to access the module specific functionality provided by the BSW Scheduler.⌋

The *Module Interlink Header* (`SchM_<Mip>.h`) defines the Basic Software Scheduler API and any associated data structures that are required by the Basic Software Scheduler implementation [9]. `BswSchedulableEntity0s` are defined in BSW Module Description Template [5].

Examples:

The CAN Driver *Module implementation* file `Can_21_EXT.c` includes the header file `SchM_Can_21_EXT.h`.

The Fee *Module implementation* file `Fee.c` includes the header file `SchM_Fee.h`.

To retrieve *Production error* `EventID` symbols and their values the *Implementation header of Diagnostic Event Manager (Dem)* is necessary:

[SWS_BSW_00008] Include *Implementation header* of *Dem*

Upstream requirements: SRS_BSW_00409

⌈If the *BSW Module* reports errors to *Dem*, then the corresponding *Implementation source* files of the *BSW Module* shall include the *Implementation header* of *Dem - Diagnostic Event Manager* (`Dem.h`).⌋

For further information, see chapter 7.2 - ***Error Handling***.

To access callbacks from other modules, the according *module headers* must be included either. It must be taken in consideration that some headers are not necessary if the usage of the according callbacks is not part of implementation after configuration. See also [SWS_BSW_00104].

[SWS_BSW_00010] Include *module headers*

Upstream requirements: SRS_BSW_00447

⌈If the *BSW Module* implementation calls *Callback functions* from other modules, then the *Implementation source* files of the *BSW Module* shall include the *module headers* from all modules defining the called *Callback functions*. *In case the callback functions are located on application layer, then the BSW module shall include the RTE exported application header file instead.*⌋

The inclusion of application header file is specified in [SWS_BSW_00023].

The implementation of *Interrupt service routines called* from *Interrupt frames* is done in the *Implementation source*. See also [SWS_BSW_00021].

[SWS_BSW_00017] Implement ISRs

Upstream requirements: SRS_BSW_00314

⌈If the *BSW Module* implements *Interrupt Service Routines*, then these routines shall be implemented in one or more of its Implementation source files.⌋

[SWS_BSW_00181] Implement ISRs in a separate file

Upstream requirements: SRS_BSW_00314

⌈If the *BSW Module* implements *Interrupt Service Routines*, then these routines should be implemented in a file or in files separated from the remaining implementation.⌋

5.1.6.2 Link time configuration source

The *Link time configuration source* contains definitions of link time configuration parameters for the *BSW Module*.

[SWS_BSW_00013] Provide *Link time configuration source* files

Upstream requirements: SRS_BSW_00346

⌈If the *BSW Module* implementation contains link time configuration parameters defined as const, the code file structure shall contain one or more files for their definition: the *Link time configuration source* files. The file names shall be formed in the following way:

`<Mip>[_<Ie>]_Lcfg.c` or `<Mip>[_<Ie>]_Cfg.c`⌋

[SWS_BSW_00014] Define all *Link time configuration parameters*

Upstream requirements: SRS_BSW_00380

⌈The *Link time configuration source* shall contain definitions for all link time configuration parameters specified for this module.⌋

See also chapter 10.2.4 - ***Link time configuration***.

5.1.6.3 Post-build time configuration source

The *Post-build time configuration source* contains definitions of post-build time configuration parameters for the *BSW Module*.

[SWS_BSW_00015] Provide *Post-build time configuration source* files

Upstream requirements: SRS_BSW_00346

⌈If the *BSW Module* implementation contains post-build time configuration parameters, then the code file structure shall contain one or more files for their definition: the *Post-build time configuration source* files. The file names shall be formed in the following way:

`<Mip>[_<Ie>]_PBcfg.c`⌋

[SWS_BSW_00063] Define all *Post-build time configuration parameters*

Upstream requirements: SRS_BSW_00380

⌈The *Post-build time configuration source* files shall contain definitions for all post-build time configuration parameters specified for this module. Definitions of Precompile and Linktime configuration parameters may as well be placed in *Post-build time configuration source* files.⌋

See also chapter 10.2.5 - ***Post-build time configuration***.

Rationale for adding Precompile and Linktime configuration parameters in *Post-build time configuration source* files:

Use Case 1: In case a new configuration container is introduced in Postbuild time all the Precompile and Linktime which may exist in this configuration container may be assigned a new value.

Use Case 2: In case a configuration container is implemented as one struct in c-code that contains at least one postbuild configurable parameter the entire struct needs to be placed in the Post-build time configuration source files.

5.1.6.4 Interrupt frame implementation source

The *Interrupt frame implementation source* contains implementation of *Interrupt frame* routines of the *BSW Module*.

The implementation of *Interrupt frames*, done within the *Interrupt frame implementation source*, is separated from the implementation of *Interrupt service routines*, which is done within the *Implementation source* ([SWS_BSW_00017])

This separation enables flexibility in the usage of different compilers and or OS integrations. For instance, the interrupt could be realized as ISR frame of the operating system or implemented directly without changing the driver code. The service routine can be called directly during module test without the need of causing an interrupt.

[SWS_BSW_00016] Provide *Interrupt frame implementation* source files

Upstream requirements: SRS_BSW_00314

⌈If the *BSW Module* implements *Interrupt frames*, then the code file structure shall contain one or more files for their implementation: the *Interrupt frame implementation source* files. The file names shall be formed in the following way:

`<Mip>[_<Ie>]_Irq.c`⌋

[SWS_BSW_00021] Implement *Interrupt frame routines*

Upstream requirements: SRS_BSW_00314

⌈The *Interrupt frame implementation source* shall contain implementation of all *Interrupt frame* routines specified for this *BSW Module*.⌋

The declaration of *Interrupt frames* routines is done in the *Implementation header*. See also [SWS_BSW_00018].

[SWS_BSW_00019] Include *Implementation Header* to *Interrupt frame implementation source*

Upstream requirements: SRS_BSW_00314

⌈The *Interrupt frame implementation source* files of a *BSW Module* shall include the *Implementation Header* of this *BSW Module*.⌋

The implementation of *Interrupt service routines* called from *Interrupt frames* is done in the *Implementation source*. See also [SWS_BSW_00017].

5.1.7 Header file structure

5.1.7.1 Implementation header

The *Implementation header* of the *BSW Module* provides the declaration of the modules' API. This header file or files are included by other modules that use the *BSW Modules*' API.

[SWS_BSW_00020] Provide *Implementation header* file

Upstream requirements: SRS_BSW_00346

⌈The header file structure shall contain one or more files that provide the declaration of functions from the *BSW Module (or Library)* API: the *Implementation header* files. The file names shall be formed in the following way:

`<Mip>[_<Ie>].h`

At least the file `<Mip>.h` shall be available.⌋

[SWS_BSW_00110] Content of *Implementation header*

Upstream requirements: SRS_BSW_00346, SRS_BSW_00302

⌈The *Implementation header* files may contain extern declarations of constants, global data and services. They shall at least contain those declarations of constants, global data and services that are available to users of the *BSW Module*.⌋

To avoid double and inconsistent definition of data types in both *BSW Module* and Software Components, common data types are defined in the *RTE Type* header file. This file is included in *BSW Module* indirectly through its *Application Types Header File*.

[SWS_BSW_00023] Include *Application Types Header File* to *Implementation header*

Upstream requirements: SRS_BSW_00447

⌈If the *BSW Module* implements AUTOSAR Services, then it shall include its *Application Types Header File* in its *Implementation header* file or files.⌋

The *Application Types Header File* is named `Rte_<swc>_Type.h`, where `<swc>` is the Short Name of the according Software Component Type. More information about this file can be found in the Specification of RTE [9] - section "Application Types Header File".

Example:

The same data Data Type `NvM_RequestResultType` is used in BSW C-API `NvM_GetErrorStatus` and in the AUTOSAR Interface `NvMService` operation `GetErrorStatus (OUT NvM_RequestResultType RequestResultPtr)`. This implies:

- The proper types shall be generated in `Rte_Type.h`.

- `Rte_Type.h` shall be included in Implementation header of BSW Module (`NvM.h`) via `Rte_NvM_Type.h`
- `Rte_Type.h` shall be included in the application types header file (`Rte_<swc>_Type.h`) of SW-C modules that are using the service `GetError Status`.

This header is included in the application header file (`Rte_<swc>.h`), which is used by the SW-C implementation. These headers are generated by the *RTE Generator*.

[SWS_BSW_00024] Include *AUTOSAR Standard Types Header* to *Implementation header*

Upstream requirements: SRS_BSW_00348

⌈If the *BSW Module* implementation uses AUTOSAR Standard Types, then its *Implementation header* file or files shall include the *AUTOSAR Standard Types Header* (`Std_Types.h`).⌋

The *AUTOSAR Standard Types Header* includes the following header:

- *Platform Specific Types Header* (`Platform_Types.h`)

For more information on *AUTOSAR Standard Types*, see also chapter 7.1.18 - ***Data types***.

[SWS_BSW_00048] Declare *API services* in *Implementation header*

Upstream requirements: SRS_BSW_00302

⌈If the *BSW Module* implements *API services*, then their declaration shall be done in its *Implementation header* file or files.⌋

See also 8.3.1 - ***General specification on API functions***.

[SWS_BSW_00254] Provide *Externals header* file

Upstream requirements: SRS_BSW_00463, SRS_BSW_00346

⌈If there are callouts that are specified or configured to be used in the *BSW Module*, then the declarations of these APIs shall be provided in `<Mip>_Externals.h`. This header should be self-contained.⌋

Example: Operating System

```
1 /* File: Os_External.h */
2 ...
3 /* Callout declaration */
4 void ErrorHook ( StatusType );
5 ...
```

Notes:

- The implementer of the callout can include `<Mip>_Externals.h` to get the declaration.
- If the implementer declares the callout on its side, it has to make sure the declaration does not differ ([SWS_BSW_00136]).
- If Post-Build variants are used, the declarations of the callouts used in the different variants shall be available in `<Mip>_Externals.h`.

[SWS_BSW_00018] Declare *Interrupt frame routines*

Upstream requirements: SRS_BSW_00314

⌈If the *BSW Module* implements *Interrupt frame routines* ([SWS_BSW_00021]), then their declaration shall be done in its *Implementation header* file or files.⌋

[SWS_BSW_00043] Declare *Interrupt Service Routines*

Upstream requirements: SRS_BSW_00439

⌈If the *BSW Module* implements *Interrupt Service Routines (ISR)*, then their declaration shall be done in its *Implementation header* file or files.⌋

[SWS_BSW_00068] Support *Interrupt Service Routines categories 1 and 2*

Upstream requirements: SRS_BSW_00439

⌈If the *BSW Module* implements *Interrupt Service Routines (ISR)* and provides declarations for both interrupt categories CAT1 and CAT2, then the interrupt category shall be selectable via configuration.⌋

See also chapter 7.1.15 - ***Interrupt service routines***.

[SWS_BSW_00210] Exclusion of MainFunction and BswModuleClientServerEntrys from the Implementation header

Upstream requirements: SRS_BSW_00415

⌈The module header files shall not include the prototype declarations of `MainFunctions` and `BswModuleClientServerEntrys` that are expected to be invoked by the *RTE/BswScheduler*.⌋

[SWS_BSW_00249] Include guard

Upstream requirements: SRS_BSW_00007

⌈All header files (except memory mapping header files) shall protect themselves against multiple inclusions in the following way:

start of file

```
#ifndef identifier
#define identifier

/* content */
```

```
5
6 #endif
```

end of file

The identifer that is used shall be unique across all header files in the project. There may be only comments outside of the `#ifndef - #endif` bracket.⌋

Example: File header.h

```
1 #ifndef HEADER_H
2 #define HEADER_H
3
4 /* implementation of header.h */
5
6 ...
7
8 /* HEADER_H */
9 #endif
```

5.1.7.2 Application Header File

If the *BSW Module* implements *AUTOSAR Services*, the according *Application Header File* is generated with the RTE. This file provides interfaces for the interaction of the *BSW Module* with the RTE. The Application Header File is named `Rte_<swc>.h`, where `<swc>` is the *Short Name* of the according *Software Component Type*.

[SWS_BSW_00025] Include *Application Header File*

Upstream requirements: SRS_BSW_00447

⌈If the *BSW Module* implements *AUTOSAR Services*, then it shall include its *Application Header File* in module files using RTE interfaces.⌋

[SWS_BSW_00069] Restrict inclusion for *Application Header File*

Upstream requirements: SRS_BSW_00447

⌈The *Application Header File* shall not be included in *BSW Module* files that are included directly or indirectly by other modules.⌋

If the *Application Header File* is included in module files which are included directly or indirectly by other modules, other Services or CDDs would also include several *Application Header Files* and this is not supported by RTE. See *Specification of RTE* [9] - section "File Contents", requirement [SWS_Rte_01006].

More information about the *Application Header File* can be found in the *Specification of RTE* [9] - section "Application Header File".

Note that the application header file includes by its own the *Application Types Header File*. See *Specification of RTE* [9], [SWS_Rte_07131], and [SWS_BSW_00023].

6 Requirements Tracing

For every *BSW Module*, both the according BSW specification and this document (*SWS BSW General*) satisfy requirements from AUTOSAR General Requirements on *Basic Software Modules* [1]. The following situations are possible:

	Requirement traceability from:		***Result for BSW Module implementation:***
	Module SWS	***SWS BSW General***	
1	"Not applicable."	"See module's SWS."	Requirement is not applicable for *BSW Module*.
2	"Not applicable."	Specified	Requirement is not applicable for *BSW Module*. The module implementation can ignore specification items from *SWS BSW General* that are tracing to this requirement. Please attempt also to comments in module's own SWS document.
3	Specified	"See module's SWS."	Requirement is applicable to *BSW Module*. The module specific SWS satisfies this requirement.
4	"Satisfied by SWS BSW General"	Specified	Requirement is applicable to *BSW Module*. SWS BSW General satisfies this requirement.
5	Specified	Specified	Requirement is applicable to *BSW Module*. Both general SWS and module specific SWS are needed to satisfy this requirement. I.e. module specific specification items complement general specification items from SWS BSW General.

Table 6.1: Requirements Traceability Options

The following tables reference the requirements specified in [1] 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_00003]	All software modules shall provide version and identification information	[SWS_BSW_00059] [SWS_BSW_00256]
[SRS_BSW_00006]	The source code of software modules above the μC Abstraction Layer (MCAL) shall not be processor and compiler dependent.	[SWS_BSW_00119]
[SRS_BSW_00007]	All Basic SW Modules written in C language shall conform to the MISRA C 2025 Standard.	[SWS_BSW_00115] [SWS_BSW_00249]
[SRS_BSW_00009]	All Basic SW Modules shall be documented according to a common standard.	[SWS_BSW_00002] [SWS_BSW_00209]
[SRS_BSW_00010]	The memory consumption of all Basic SW Modules shall be documented for a defined configuration for all supported platforms.	[SWS_BSW_00002]

▽

△

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_BSW_00150]
[SRS_BSW_00159]	All modules of the AUTOSAR Basic Software shall support a tool based configuration	[SWS_BSW_00116]
[SRS_BSW_00160]	Configuration files of AUTOSAR Basic SW module shall be readable for human beings	[SWS_BSW_00157]
[SRS_BSW_00164]	The Implementation of interrupt service routines shall be done by the Operating System, complex drivers or modules	[SWS_BSW_00137] [SWS_BSW_00182]
[SRS_BSW_00167]	All AUTOSAR Basic Software Modules shall provide configuration rules and constraints to enable plausibility checks	[SWS_BSW_00061]
[SRS_BSW_00171]	Optional functionality of a Basic-SW component that is not required in the ECU shall be configurable at pre-compile-time	[SWS_BSW_00029]
[SRS_BSW_00300]	Unique Name of Basic Software Modules	[SWS_BSW_00101] [SWS_BSW_00102] [SWS_BSW_00103]
[SRS_BSW_00301]	All AUTOSAR Basic Software Modules shall only import the necessary information	[SWS_BSW_00104]
[SRS_BSW_00302]	All AUTOSAR Basic Software Modules shall only export information needed by other modules	[SWS_BSW_00048] [SWS_BSW_00060] [SWS_BSW_00105] [SWS_BSW_00110] [SWS_BSW_00200]
[SRS_BSW_00304]	All AUTOSAR Basic Software Modules shall use only AUTOSAR data types instead of native C data types	[SWS_BSW_00120] [SWS_BSW_00209]
[SRS_BSW_00305]	Data types naming convention	[SWS_BSW_00146]
[SRS_BSW_00306]	AUTOSAR Basic Software Modules shall be compiler and platform independent	[SWS_BSW_00121]
[SRS_BSW_00307]	Global variables naming convention	[SWS_BSW_00130]
[SRS_BSW_00308]	AUTOSAR Basic Software Modules shall not define global data in their header files, but in the C file	[SWS_BSW_00129]
[SRS_BSW_00309]	All AUTOSAR Basic Software Modules shall indicate all global data with read-only purposes by explicitly assigning the const keyword	[SWS_BSW_00131]
[SRS_BSW_00312]	Shared code shall be reentrant	[SWS_BSW_00190] [SWS_BSW_00191] [SWS_BSW_00192]
[SRS_BSW_00314]	All internal driver modules shall separate the interrupt frame definition from the service routine	[SWS_BSW_00016] [SWS_BSW_00017] [SWS_BSW_00018] [SWS_BSW_00019] [SWS_BSW_00021] [SWS_BSW_00181]
[SRS_BSW_00318]	Each AUTOSAR Basic Software Module file shall provide version numbers in the header file	[SWS_BSW_00059] [SWS_BSW_00256]

▽

△

Requirement	Description	Satisfied by
[SRS_BSW_00321]	The version numbers of AUTOSAR Basic Software Modules shall be enumerated according specific rules	[SWS_BSW_00162]
[SRS_BSW_00323]	All AUTOSAR Basic Software Modules shall check passed API parameters for validity	[SWS_BSW_00049] [SWS_BSW_00203]
[SRS_BSW_00325]	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	[SWS_BSW_00167]
[SRS_BSW_00327]	Error values naming convention	[SWS_BSW_00125]
[SRS_BSW_00328]	All AUTOSAR Basic Software Modules shall avoid the duplication of code	[SWS_BSW_00127]
[SRS_BSW_00333]	For each callback function it shall be specified if it is called from interrupt context or not	[SWS_BSW_00167]
[SRS_BSW_00334]	Machine readable module description	[SWS_BSW_00001] [SWS_BSW_00208] [SWS_BSW_00238]
[SRS_BSW_00335]	Status values naming convention	[SWS_BSW_00124]
[SRS_BSW_00336]	Basic SW module shall be able to shutdown	[SWS_BSW_00232] [SWS_BSW_00233]
[SRS_BSW_00337]	Classification of development errors	[SWS_BSW_00042] [SWS_BSW_00045] [SWS_BSW_00073] [SWS_BSW_00201] [SWS_BSW_00212] [SWS_BSW_00243] [SWS_BSW_00255] [SWS_BSW_00264]
[SRS_BSW_00339]	Reporting of production relevant error status	[SWS_BSW_00046] [SWS_BSW_00066] [SWS_BSW_00207] [SWS_BSW_00208]
[SRS_BSW_00341]	Module documentation shall contains all needed informations	[SWS_BSW_00003]
[SRS_BSW_00342]	It shall be possible to create an AUTOSAR ECU out of modules provided as source code and modules provided as object code, even mixed	[SWS_BSW_00117]
[SRS_BSW_00344]	BSW Modules shall support link-time configuration	[SWS_BSW_00056] [SWS_BSW_00226]
[SRS_BSW_00345]	BSW Modules shall support pre-compile configuration	[SWS_BSW_00226]
[SRS_BSW_00346]	All AUTOSAR Basic Software Modules shall provide at least a basic set of module files	[SWS_BSW_00004] [SWS_BSW_00005] [SWS_BSW_00013] [SWS_BSW_00015] [SWS_BSW_00020] [SWS_BSW_00110] [SWS_BSW_00254]
[SRS_BSW_00347]	A Naming seperation of different instances of BSW drivers shall be in place	[SWS_BSW_00102] [SWS_BSW_00126] [SWS_BSW_00153]
[SRS_BSW_00348]	All AUTOSAR standard types and constants shall be placed and organized in a standard type header file	[SWS_BSW_00024]
[SRS_BSW_00353]	All type definitions of target and compiler specific scope shall be placed and organized in a single type header	[SWS_BSW_00120]

▽

△

Requirement	Description	Satisfied by
[SRS_BSW_00360]	AUTOSAR Basic Software Modules callback functions are allowed to have parameters	[SWS_BSW_00173]
[SRS_BSW_00373]	The main processing function of each AUTOSAR Basic Software Module shall be named according the defined convention	[SWS_BSW_00153] [SWS_BSW_00154]
[SRS_BSW_00374]	All Basic Software Modules shall provide a readable module vendor identification	[SWS_BSW_00059] [SWS_BSW_00161] [SWS_BSW_00256]
[SRS_BSW_00378]	AUTOSAR shall provide a boolean type	[SWS_BSW_00142]
[SRS_BSW_00379]	All software modules shall provide a module identifier in the header file and in the module XML description file.	[SWS_BSW_00059] [SWS_BSW_00256]
[SRS_BSW_00380]	Configuration parameters being stored in memory shall be placed into separate c-files	[SWS_BSW_00014] [SWS_BSW_00063] [SWS_BSW_00227]
[SRS_BSW_00385]	List possible error notifications	[SWS_BSW_00144] [SWS_BSW_00205] [SWS_BSW_00222]
[SRS_BSW_00396]	The Basic Software Module specifications shall specify the supported configuration classes for changing values and multiplicities for each parameter/container	[SWS_BSW_00237]
[SRS_BSW_00397]	The configuration parameters in pre-compile time are fixed before compilation starts	[SWS_BSW_00183]
[SRS_BSW_00398]	The link-time configuration is achieved on object code basis in the stage after compiling and before linking	[SWS_BSW_00184]
[SRS_BSW_00399]	Parameter-sets shall be located in a separate segment and shall be loaded after the code	[SWS_BSW_00227]
[SRS_BSW_00400]	Parameter shall be selected from multiple sets of parameters after code has been loaded and started	[SWS_BSW_00050] [SWS_BSW_00226] [SWS_BSW_00228]
[SRS_BSW_00402]	Each module shall provide version information	[SWS_BSW_00059] [SWS_BSW_00256]
[SRS_BSW_00404]	BSW Modules shall support post-build configuration	[SWS_BSW_00160] [SWS_BSW_00226]
[SRS_BSW_00405]	BSW Modules shall support multiple configuration sets	[SWS_BSW_00226] [SWS_BSW_00228]
[SRS_BSW_00406]	API handling in uninitialized state	[SWS_BSW_00230] [SWS_BSW_00231]
[SRS_BSW_00408]	All AUTOSAR Basic Software Modules configuration parameters shall be named according to a specific naming rule	[SWS_BSW_00126]
[SRS_BSW_00409]	All production code error ID symbols are defined by the Dem module and shall be retrieved by the other BSW modules from Dem configuration	[SWS_BSW_00008] [SWS_BSW_00143]
[SRS_BSW_00410]	Compiler switches shall have defined values	[SWS_BSW_00123]

▽

△

Requirement	Description	Satisfied by
[SRS_BSW_00414]	Init functions shall have a pointer to a configuration structure as single parameter	[SWS_BSW_00049] [SWS_BSW_00050] [SWS_BSW_00227]
[SRS_BSW_00415]	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	[SWS_BSW_00007] [SWS_BSW_00210]
[SRS_BSW_00423]	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	[SWS_BSW_00001] [SWS_BSW_00040]
[SRS_BSW_00424]	BSW module main processing functions shall not be allowed to enter a wait state	[SWS_BSW_00156]
[SRS_BSW_00426]	BSW Modules shall ensure data consistency of data which is shared between BSW modules	[SWS_BSW_00001] [SWS_BSW_00038] [SWS_BSW_00134] [SWS_BSW_00265] [SWS_BSW_00266]
[SRS_BSW_00427]	ISR functions shall be defined and documented in the BSW module description template	[SWS_BSW_00001] [SWS_BSW_00041] [SWS_BSW_00065]
[SRS_BSW_00428]	A BSW module shall state if its main processing function(s) has to be executed in a specific order or sequence	[SWS_BSW_00054]
[SRS_BSW_00429]	Access to OS is restricted	[SWS_BSW_00138] [SWS_BSW_00257]
[SRS_BSW_00433]	Main processing functions are only allowed to be called from task bodies provided by the BSW Scheduler	[SWS_BSW_00133]
[SRS_BSW_00437]	Memory mapping shall provide the possibility to define RAM segments which are not to be initialized during startup	[SWS_BSW_00006]
[SRS_BSW_00438]	Configuration data shall be defined in a structure	[SWS_BSW_00050] [SWS_BSW_00057] [SWS_BSW_00158]
[SRS_BSW_00439]	Enable BSW modules to handle interrupts	[SWS_BSW_00043] [SWS_BSW_00068]
[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_BSW_00180]
[SRS_BSW_00441]	Naming convention for type, macro and function	[SWS_BSW_00124]
[SRS_BSW_00447]	Standardizing Include file structure of BSW Modules Implementing Autosar Service	[SWS_BSW_00010] [SWS_BSW_00023] [SWS_BSW_00025] [SWS_BSW_00069] [SWS_BSW_00147]
[SRS_BSW_00450]	A Main function of a un-initialized module shall return immediately	[SWS_BSW_00037] [SWS_BSW_00071] [SWS_BSW_00072]
[SRS_BSW_00451]	Hardware registers shall be protected if concurrent access to these registers occur	[SWS_BSW_00179] [SWS_BSW_00188]
[SRS_BSW_00452]	Classification of runtime errors	[SWS_BSW_00219] [SWS_BSW_00222]
[SRS_BSW_00457]	Callback functions of Application software components shall be invoked by the Basis SW	[SWS_BSW_00218]
[SRS_BSW_00458]	Classification of production errors	[SWS_BSW_00204] [SWS_BSW_00205]

▽

△

Requirement	Description	Satisfied by
[SRS_BSW_00459]	It shall be possible to concurrently execute a service offered by a BSW module in different partitions	[SWS_BSW_00191]
[SRS_BSW_00463]	Naming convention of callout prototypes	[SWS_BSW_00136] [SWS_BSW_00254]
[SRS_BSW_00464]	File names shall be considered case sensitive regardless of the filesystem in which they are used	[SWS_BSW_00170]
[SRS_BSW_00465]	It shall not be allowed to name any two files so that they only differ by the cases of their letters	[SWS_BSW_00171]
[SRS_BSW_00467]	The init / deinit services shall only be called by BswM or EcuM	[SWS_BSW_00150] [SWS_BSW_00152]
[SRS_BSW_00470]	Execution frequency of production error detection	[SWS_BSW_00206]
[SRS_BSW_00477]	The functional interfaces of AUTOSAR BSW modules shall be specified in C99	[SWS_BSW_00234]
[SRS_BSW_00480]	Null pointer errors shall follow a naming rule	[SWS_BSW_00212]
[SRS_BSW_00483]	BSW Modules shall handle buffer alignments internally	[SWS_BSW_00241]
[SRS_BSW_00485]	Input parameters of structure type shall be passed as a reference to a constant structure	[SWS_BSW_00186]
[SRS_BSW_00486]	Input parameters of array type shall be passed as a reference to the constant array base type	[SWS_BSW_00186]
[SRS_BSW_00487]	Errors for module initialization shall follow a naming rule	[SWS_BSW_00243]
[SRS_BSW_00488]	Classification of security events	[SWS_BSW_00244]
[SRS_BSW_00489]	Reporting of security events	[SWS_BSW_00250] [SWS_BSW_00258]
[SRS_BSW_00493]	Definition of security event ID symbols	[SWS_BSW_00251]
[SRS_BSW_00496]	BSW modules shall be able to handle PDU meta data	[SWS_BSW_00239] [SWS_BSW_00240] [SWS_BSW_00241] [SWS_BSW_00242] [SWS_BSW_00245]
[SRS_BSW_00497]	Handling of corrupted NV data.	[SWS_BSW_00263]
[SRS_BSW_00498]	Errors for module double initialization shall follow a naming rule	[SWS_BSW_00264]

Table 6.2: Requirements Tracing

7 Functional specification

7.1 General implementation specification

7.1.1 Conformance to MISRA C and C standard

MISRA C describes programming rules for the C programming language and a process to implement and follow these rules.

[SWS_BSW_00115] Conformance to MISRA C

Upstream requirements: SRS_BSW_00007

⌈If the *BSW Module* implementation is written in C language, then it shall conform to the MISRA C 2025 Standard [10].⌋

Only in technically reasonable and exceptional cases, a MISRA violation is permissible. Such violations against MISRA rules shall be clearly identified and documented within comments in the C source code.

Example: MISRA violations could be commented next to the instruction causing the violation saying `/* MRYY RULE XX VIOLATION: This is the reason why the MISRA rule could not be followed in this special case*/` while YY is two digit year representation of the MISRA version and XX is the MISRA number. For MISRA directives violation the following comment could be used: `/* MR12 DIR XX VIOLATION: This is the reason why the MISRA directive was not be followed in this case*/`

[SWS_BSW_00234] Service Interface Conformance to C standard

Upstream requirements: SRS_BSW_00477

⌈The external interface binding of the *BSW Modules* shall be conform to ISO/IEC 9899:1999 (C99) standard. The number of significant characters allowed for external identifier can exceed the specified minimum (31 characters).⌋

7.1.2 Conformance to AUTOSAR Basic Software Requirements

The *BSW Module* implementation shall conform to all applicable *Basic Software Requirements*, which are described in document SRS BSW General [1].

Note that some *BSW Module* specifications, in particular included code examples, may ignore some General BSW requirement for sake of simplicity. Examples:

- Memory abstraction is not used within the BSW specification text because of readability.
- The use of pre-processor directives (`#defines`) without "u" or "s" is widely present in the specifications, but this violates MISRA.

However, the implementation shall not interpret this as a simplification, redefinition or relaxation of general BSW requirements.

7.1.3 Conformance to AUTOSAR Methodology

The *BSW Module* implementation shall consider the AUTOSAR (see chapter 3.1):, e.g. supporting the capability use cases *Develop Basic Software and Integrate Software for ECU*.

[SWS_BSW_00116] Support to tool-based configuration

Upstream requirements: SRS_BSW_00159

⌈The *BSW Module* implementation shall support a tool based configuration, as described in AUTOSAR Methodology [7].⌋

For more information about ECU configuration, see also AUTOSAR Specification of ECU Configuration [6].

With the AUTOSAR Methodology it is possible to configure an AUTOSAR ECU out of *BSW Modules* provided as source code and out of *BSW Modules* provided as object code, or even mixed. This must be of course supported by the implementation, i.e. it shall not require that the source code is always part of the delivery.

[SWS_BSW_00117] Support object code delivery and configuration

Upstream requirements: SRS_BSW_00342

⌈The *BSW Module* implementation shall support configuration of its link-time and post-build configuration parameters even if only the object code and the corresponding header files are available, i.e. even if the source code files are not available.⌋

7.1.4 Platform independency

According to their dependency on implementation platform, this specification classifies *BSW Modules* in two distinct categories:

- *Platform independent BSW Modules*: All *BSW Modules* except *Complex Drivers*, *MCAL* modules and the *OS*.
- *Platform dependent BSW Modules*: *MCAL* modules, *Complex Drivers*, *OS*.

The platform dependency comprises dependencies on used toolchain and hardware, e.g. compiler and processor dependencies

Platform dependent BSW Modules have or may have direct access to microcontroller hardware. Thus, their implementation is platform specific.

Platform independent BSW Modules can be developed once and then be compilable for all platforms without any changes. Any necessary processor or compiler specific

instructions (e.g. memory locators, pragmas, use of atomic bit manipulations etc.) have to be encapsulated by macros and imported through *include* files. This is necessary to minimize number of variants and the according development effort.

The *Microcontroller Abstraction Layer (MCAL)* is defined in *AUTOSAR Layered Software Architecture* [4]. The list of *BSW Modules* from *MCAL* is available in Chapter 11 "List of Basic Software Modules": Microcontroller Drivers, I/O Drivers, Communication Drivers and Memory Drivers.

[SWS_BSW_00119] Platform independent BSW Modules

Upstream requirements: SRS_BSW_00006

⌈If the *BSW Module* is classified as *Platform independent BSW Module*, then its source code shall not be processor dependent.⌋

The direct use of not standardized keywords like `_near`, `_far`, `_pascal` in the source code would create compiler and platform dependencies, that must strictly be avoided. If no precautions are made, portability and reusability of affected code is deteriorated and effective release management is costly and hard to maintain.

[SWS_BSW_00121] Usage of platform or compile specific keywords is restricted

Upstream requirements: SRS_BSW_00306

⌈The *BSW Module* implementation shall not use compiler and platform specific keywords directly.⌋

7.1.5 Configurability

Plausibility checks on configuration parameters can be made by a configuration tool during configuration or by the pre-processor during runtime. See also [SWS_BSW_00061]

Detailed configuration rules and constraints may also be part of module's own specification and the *BSW Module's documentation*, which is delivered with the module implementation.

Optional functionalities of a *BSW Module* shall not consume resources (RAM, ROM and runtime). These functionalities can be enabled or disabled at pre-compile time with suitable configuration parameters, like defined in chapter 10 of the respective *BSW Module* specification.

[SWS_BSW_00029] Implement configuration of optional functionality

Upstream requirements: SRS_BSW_00171

⌈If the *BSW Module* contains optional functionality, then this functionality shall be enabled (`STD_ON`) or disabled (`STD_OFF`) by a *Pre-compile time configuration parameter*.⌋

Disabled functionality will not become part of compiled code. If the code is automatically generated, e.g. after configuration, the disabled functionality may even not be part of source code. It may also never have been implemented, if the BSW software provider does not support this configuration.

These symbols, `STD_ON` and `STD_OFF`, and their values are defined in `Std_Types.h` [SWS_BSW_00024].

The module configuration shall be according to the AUTOSAR Methodology [7], see [SWS_BSW_00024]. The module configuration parameters are defined in chapter 10 of the corresponding *BSW Module* specification.

[SWS_BSW_00123] Check compiler switches by comparison with defined values

Upstream requirements: SRS_BSW_00410

⌈Compiler switches shall be compared with defined values. Simply checking if a compiler switch is defined shall not be used in implementation.⌋

Example:

```
#if (EEP_21_LDEXT_DEV_ERROR_DETECT == STD_ON )
```

...

Example of a wrong implementation:

```
#ifdef EEP_21_LDEXT_DEV_ERROR_DETECT
```

...

7.1.6 Various naming conventions

[SWS_BSW_00124] Naming convention for enumeration literals, status values and pre-processor directives

Upstream requirements: SRS_BSW_00441, SRS_BSW_00335

⌈All enumeration literals, status values and pre-processor directives (`#define`) shall be labeled in the following way:

`<MIP>_<SN>`

Where here `<MIP>` is the *Capitalized module implementation prefix* of this *BSW Module* [SWS_BSW_00102] and `<SN>` is the specific name. Only capital letters shall be used. If <SN> consists of several words, they shall be separated by underscore.

The pre-processor directives `E_OK` and `E_NOT_OK` are exceptions to this rule.⌋

Example: The Eeprom driver has the following status values:

```
EEP_21_LDEXT_UNINIT
EEP_21_LDEXT_IDLE
EEP_21_LDEXT_BUSY
```

Examples for pre-processor directives:

```
#define EEP_21_LDEXT_PARAM_CONFIG
#define EEP_21_LDEXT_SIZE
```

Example for enumeration literals:

```
typedef enum
{
  EEP_21_LDEXT_PARAM_CONFIG = 0,
  EEP_21_LDEXT_ARE = 1,
  EEP_21_LDEXT_EV = 2
} Eep_21_LDExt_Notification_Type;
```

[SWS_BSW_00125] Naming convention for Error values

Upstream requirements: SRS_BSW_00327

⌈Error values shall be named in the following way:

<MIP_E_<EN>

Where here <MIP> is the *Capitalized module implementation prefix* of this *BSW Module* [SWS_BSW_00102] <EN> is the error name. Only capital letters shall be used. If <EN> consists of several words, they shall be separated by underscore.⌋

Example: The EEPROM driver has the following error values:

```
EEP_21_LDEXT_E_BUSY
EEP_21_LDEXT_E_PARAM_ADDRESS
EEP_21_LDEXT_E_PARAM_LENGTH
EEP_21_LDEXT_E_WRITE_FAILED
```

7.1.7 Configuration parameters

The *BSW Module* implementation must use *Configuration parameter names*. For further information, see also chapter 10.2.2.

[SWS_BSW_00126] Naming conventions for *Configuration parameters names*

Upstream requirements: SRS_BSW_00408, SRS_BSW_00347

⌈*Configuration parameters* shall be named in one of the following ways:

Camel case: <Ma><Pn>
Upper case: <MA><PN>

Where:

- <Pn> is the specific parameter name in camel case;
- <PN> is the specific parameter name in upper case;

The term <Pn> (or <PN>) may consist of several words which may or may not be separated by underscore.⌋

Example:

- `CanIfTxConfirmation`

7.1.8 Shared code

Duplicated code may result in bugs during code maintenance. This can be avoided by sharing code whenever necessary. Shared code eases functional composition, reusability, code size reduction and maintainability.

[SWS_BSW_00127] Avoid duplication of code

Upstream requirements: SRS_BSW_00328

⌈The *BSW Module* implementation shall avoid duplication of code.⌋

Note that if the *BSW Module* implements shared code, then the implementation may need to ensure reentrancy for this code if it is exposed to preemptive environments. Reentrancy support is part of the API specification. See also chapter 8.3.1.

7.1.9 Global data

To avoid multiple definition and uncontrolled spreading of global data, the visibility of global variables must be limited.

[SWS_BSW_00129] Definition of global variables

Upstream requirements: SRS_BSW_00308

⌈If the *BSW Module* defines global variables, then their definition shall take place in the *Implementation source* file.⌋

[SWS_BSW_00130] Naming convention for global variables

Upstream requirements: SRS_BSW_00307

⌈All global variables defined by the BSW Module shall be labeled according to the following:

`<Mip>_<Vn>`

Where `<Mip>` is the *Module implementation prefix* of the *BSW Module* [SWS_BSW_00102] and `<Vn>` is the *Variable name*, which shall be written in *camel case*.⌋

Example of global variable names:

- `Can_21_Ext_MessageBuffer[CAN_21_EXT_BUFFER_LENGTH]`
- `Nm_RingData[NM_RINGDATA_LENGTH]`

In principle, all global data shall be avoided due to extra blocking efforts when used in preemptive runtime environments. Unforeseen effects may occur if no precautions

were made. If data is intended to serve as constant data, global exposure is permitted only if data is explicitly declared read-only using the `const` qualifier.

[SWS_BSW_00131] Definition of constant global variables

Upstream requirements: SRS_BSW_00309

⌈If the *BSW Module* defines global variables with read-only purpose, this shall be formalized by assigning the `const` qualifier to their definitions and declarations.⌋

7.1.10 Usage of macros and inline functions

The usage of macros and inline functions instead of functions is allowed to improve the runtime behavior. Special attention has to be paid with regard to reentrant functions.

Macros can be used instead of functions where source code is used and runtime is critical. Inline functions can be used for the same purpose. Inline functions have the advantage (compared to macros) that the compiler can do type checking of function parameters and return values.

7.1.11 Calling Scheduled functions (Main processing functions)

Main Processing Functions, also called *Scheduled Functions*, are defined in chapter 8.4.

To avoid indirect and non-transparent timing dependencies between *BSW Modules*, the calling of *Scheduled functions* is restricted to task bodies provided by the *BSW Scheduler* - see the Specification of RTE [9].

[SWS_BSW_00133] Calling *Scheduled functions* is restricted

Upstream requirements: SRS_BSW_00433

⌈The *BSW Module* implementation shall not contain calls to *Scheduled functions (Main processing functions)*.⌋

Calling *Scheduled functions* of an un-initialized *BSW Module* may result in undesired and non-defined behavior.

[SWS_BSW_00037] Behavior of un-initialized *Scheduled functions*

Upstream requirements: SRS_BSW_00450

⌈If a *Scheduled functions (Main processing functions)* of un-initialized *BSW Module* is called from the *BSW Scheduler*, then it shall return immediately without performing any functionality and without raising any errors.⌋

7.1.12 Exclusive areas

Exclusive areas are defined to allow priority determination for preventing simultaneous access to shared resources. Every *Exclusive area* has a unique name. The description of *Exclusive areas* includes the accessing *Scheduled functions (Main processing functions)*, API services, *Callback functions* and ISR functions.

[SWS_BSW_00038] Define and document *Exclusive areas*

Upstream requirements: SRS_BSW_00426

⌈The *Exclusive areas* of the *BSW Module shall* be defined and documented as described in the specification of *BSW Module Description Template* [5] within the *BSW Module Description*.⌋

[SWS_BSW_00134] Restriction to usage of *Exclusive areas*

Upstream requirements: SRS_BSW_00426

⌈The *Exclusive areas* of the *BSW Module* shall only protect module internal data.⌋

7.1.13 Callouts

The memory segment used for a *Callout function* is not known to the module developer. The integrator needs the freedom to map these functions independently from the module design. Refer to Chapter 7.4.4 of the *Memory Mapping* [8] document for additional details.

[SWS_BSW_00136] Memory section and memory class of *Callout functions*

Upstream requirements: SRS_BSW_00463

⌈Each *Callout function* shall be mapped to its own memory section.⌋

For example:

```
#define COM_START_SEC_CODE_SOMEMODULE_SOMECALLOUT
#include "Com_MemMap.h"

void Somemodule_Some_Callout (void);

#define COM_STOP_SEC_CODE_SOMEMODULE_SOMECALLOUT
#include "Com_MemMap.h"
```

7.1.14 AUTOSAR Interfaces

AUTOSAR Services are located in the BSW, but have to interact with AUTOSAR Software Components above the RTE via ports, which realize *AUTOSAR Interfaces*. Therefore, the RTE generator shall be able to read the interface description to generate the RTE properly.

[SWS_BSW_00040] Define and document implemented AUTOSAR Interfaces

Upstream requirements: SRS_BSW_00423

⌈If the *BSW Module* implements *AUTOSAR Services*, then the related *AUTOSAR Interfaces* shall be defined and documented as described in the specification of *Software Component Template* [11] within the *BSW Module Description*.⌋

Note that the *BSW Module Description Template* inherits the description classes from the *Software Component Template*.

7.1.15 Interrupt service routines

The implementation of *Interrupt Service Routines (ISR)* is highly microcontroller dependent. See also chapter 7.1.4.

[SWS_BSW_00137] ISR implementation is platform dependent

Upstream requirements: SRS_BSW_00164

⌈If the *BSW Module* is classified as *Platform independent BSW Module*, it shall not implement interrupt service routines.⌋

For more explanation on *Platform independent BSW Modules*, see section 7.1.4.

[SWS_BSW_00167] Keep runtime of ISR as short as possible

Upstream requirements: SRS_BSW_00325, SRS_BSW_00333

⌈The runtime of *Interrupt Service Routines (ISR)* and functions that are running in interrupt context should be kept short. This affects also, for instance, *Callback functions* which are called from ISRs.

Where an *ISR* is likely to take a long time, an *Operating System* task should be used instead.⌋

ISR functions are defined with a name and the category according to the AUTOSAR OS, see [12].

[SWS_BSW_00041] Define and document ISR routines

Upstream requirements: SRS_BSW_00427

⌈If the *BSW Module* implements *Interrupt service routines (ISR)*, then these functions shall be defined and documented as described in the specification of *BSW Module Description Template* [5] within the *BSW Module Description*.⌋

[SWS_BSW_00065] Support for interrupt category CAT2

Upstream requirements: SRS_BSW_00427

⌈If the *BSW Module* implements *Interrupt service routines (ISR)*, then the implementation shall at least support interrupt category CAT2.⌋

The AUTOSAR architecture does not allow execution in interrupt context on application level. Considering this, special care is needed with nested functions called by interrupt routines.

[SWS_BSW_00182] The transition from ISR to OS task is restricted

Upstream requirements: SRS_BSW_00164

⌈If the *BSW Module* has implementation of *Interrupt Service Routines (ISR)* and a transition from an *ISR* to an *OS task* is needed, then this transition shall take place at the lowest level possible of the Basic Software:

- In the case of *CAT2 ISR* this shall be at the latest in the *RTE*.
- In the case of *CAT1 ISR* this shall be at the latest in the *MCAL layer*.

⌋

The definition of ISR categories CAT1 and CAT2 is available in AUTOSAR General Requirements on Basic Software Modules [1]. For more information see also the Specification of RTE [9] Chapter 4.2.5 *"Interrupt decoupling and notifications"*.

A *BSW Module* that handles interrupts shall be delivered partially or completely as source code so that it can be compiled to use CAT1 or CAT2 interrupts. See also [SWS_BSW_00043].

Example: A *BSW Module* from MCAL layer is delivered as object code. The interrupt handler could be written as a pair of small stubs (a CAT1 stub and a CAT2 stub) that are delivered as source code. During the module integration the code is compiled as necessary - the main handler is called.

7.1.16 Restricted OS functionality access

To avoid too much complexity in the OS integration of *BSW Modules*, some restrictions in the usage of OS services are necessary.

[SWS_BSW_00138] Restriction to usage of OS services

Upstream requirements: SRS_BSW_00429

⌈The *BSW Module* implementation is only allowed to use OS services according to the following table:

See Table in [SWS_BSW_00257].⌋

[SWS_BSW_00257]

Upstream requirements: SRS_BSW_00429

⌈

OS Services	RTE , BSW Sched-uler, BswM, CDD	EcuM	MCAL	StbM	Other BSW Modules
Activate Task	√				
Terminate Task	√				
Chain Task	√				
Schedule	√				
GetTaskID	√				
GetTaskState	√				
DisableAllInterrupts	√	√			
EnableAllInterrupts	√	√			
SuspendAllInterrupts	√		√		
ResumeAllInterrupts	√		√		
SuspendOSInterrupts	√		√		
ResumeOSInterrupts	√		√		
GetResource	√	√			
ReleaseResource	√	√			
SetEvent	√				
ClearEvent	√				
GetEvent	√				
WaitEvent	√				
GetAlarmBase	√				
GetAlarm	√				
SetRelAlarm	√				
SetAbsAlarm	√				
CancelAlarm	√				
GetActiveApplicationMode	√	√			
StartOS		√			
ShutdownOS		√			
GetApplicationID	√	√	√	√	√
StartScheduleTableRel	√	√			
StartScheduleTableAbs	√	√			
StartScheduleTableSynchron	√	√			
StopScheduleTable	√	√			
NextScheduleTable	√	√			
SyncScheduleTable	√	√		√	
GetScheduleTableStatus	√	√		√	
SetScheduleTableAsync	√	√			
IncrementCounter	√				

▽

△

GetCounterValue	√	√	√	√	√
GetElapsedValue	√	√	√	√	√
TerminateApplication	√				
AllowAccess	√				
GetApplicationState	√				
GetCurrentApplicationID	√	√	√	√	√
GetNumberOfActivatedCores	√				
GetCoreID	√	√	√	√	√
StartCore		√			
GetSpinlock	√	√	√		
ReleaseSpinlock	√	√	√		
TryToGetSpinlock	√	√	√		
ShutdownAllCores		√			
ReadPeripheral8	√		√		
ReadPeripheral16	√		√		
ReadPeripheral32	√		√		
WritePeripheral8	√		√		
WritePeripheral16	√		√		
WritePeripheral32	√		√		
ModifyPeripheral8	√		√		
ModifyPeripheral16	√		√		
ModifyPeripheral32	√		√		
DisableInterruptSource	√		√		
EnableInterruptSource	√		√		
ClearPendingInterrupt	√		√		
ActivateTaskAsyn	√				
SetEventAsyn	√				
GetISRID	√		√		
CallTrustedFunction	√	√	√	√	√
CheckISRMemoryAccess	√	√	√	√	√
CheckTaskMemoryAccess	√	√	√	√	√
CheckObjectAccess	√	√	√	√	√
CheckObjectOwnership	√	√	√	√	√
BudgetReplenish	√				

⌋

The according services are described in AUTOSAR OS.

Note:

The table above does not contain the *IOC* services of the *OS* as they are only used by the *Rte*.

7.1.17 Access to hardware registers

[SWS_BSW_00179] Concurrent access to registers

Upstream requirements: SRS_BSW_00451

⌈All *BSW modules* with direct access to hardware registers shall tolerate concurrent access to these registers from other modules, especially from Complex Drivers. This is required for the following registers:

- registers which are currently not used due to configuration reasons, e.g. channel or group not configured/enabled
- common registers with fields or bits which are used widely, e.g. interrupt mask, memory protection bits

BSW modules shall tolerate concurrent access to HW registers using defensive behavior and the techniques like:

- Protecting the read-modify-write access from interruption
- Using atomic (non-interruptible) instructions for read-modify-write access
- Protecting the access to set of registers, which have to be modified together, from interruption

⌋

Note:

- Memory mapped hardware registers in multi-master systems (multi-core systems, systems with DMA) are assumed to be manipulated by one master only
- Memory mapped hardware registers are not assumed to be manipulated by the non-maskable interrupt routines or non-maskable exception/trap routines

[SWS_BSW_00188] Access to "write-once" registers

Upstream requirements: SRS_BSW_00451

⌈If a MCAL driver initializes "write-once" registers, then the driver shall offer configuration options to disable the functionalities that have access those register, or have dependencies to them.⌋

Example:

In MCU, there should be a switch to disable the call to `Mcu_InitClock()`, if the clock set-up is performed during the start-up code, before AUTOSAR platform is started and the hardware does not allow reconfiguration.

7.1.18 Data types

7.1.18.1 AUTOSAR Standard Types

All AUTOSAR standard types and constants are placed and organized in the *AUTOSAR Standard Types Header* (`Std_Types.h`). This header:

- includes the *Platform Specific Types Header* (`Platform_Types.h`)
- defines the type `Std_ReturnType`
- defines `E_OK` and `E_NOT_OK` symbols and their values
- defines `STD_ON` and `STD_OFF` symbols and their values

See also [SWS_BSW_00024].

7.1.18.2 Platform Specific Types

Changing the microcontroller and or compiler shall only affect a limited number of files. Thus in AUTOSAR all integer type definitions of target and compiler specific scope are placed and organized in a single file, the *Platform Specific type header* (`Platform_Types.h`).

See also the *Specification of Platform Types* [13].

7.1.18.2.1 AUTOSAR Integer Data Types

The usage of native C-data types (`char`, `int`, `short`, `long`) is in general not portable and reusable throughout different platforms.

[SWS_BSW_00120] Do not use native C data types

Upstream requirements: SRS_BSW_00304, SRS_BSW_00353

⌈The *BSW Module* shall not use native C data types. AUTOSAR Integer Data Types shall be used instead. These types are defined in the *Platform Specific Types Header* (`Platform_Types.h`)⌋

The *Platform Specific Types Header* (`Platform_Types.h`) is included through the AUTOSAR Standard Types Header (`Std_Types.h`). See [SWS_BSW_00024].

The following AUTOSAR Integer Data Types are available:

- Fixed size guaranteed:

Data type	Range
uint8	0..255
uint16	0..65535
uint32	0..4294967295
uint64	0..18446744073709551615
sint8	-128..127
sint16	-32768..32767
sint32	-2147483648..2147483647
sint64	-9223372036854775808..9223372036854775807

- Minimum size guaranteed, best type is chosen for specific platform (only allowed for module internal use, not for API parameters)

Data type	Range
uint8_least	At least uint8
uint16_least	At least uint16
uint32_least	At least uint32
sint8_least	At least sint8
sint16_least	At least sint16
sint32_least	At least sint32

The data types with suffix _least can be chosen if optimal performance is required (e.g. for loop counters).

Example: Both uint8_least and uint32_least could be compiled as 32 bit on a 32 bit platform.

Hint: For integer variables without restricted value ranges the AUTOSAR integer types defined in Platform_Types.h should be used.

7.1.18.2.2 Boolean type

For simple logical values, for their checks and for API return values the AUTOSAR type boolean, defined in Platform_Types.h, can be used. For usage with this type, the following values are also defined: TRUE, FALSE.

[SWS_BSW_00142] Allowed operations with boolean variables

Upstream requirements: SRS_BSW_00378

⌈The only allowed operations with variables from type boolean are: assignment, return and test for equality, inequality and logical not with TRUE or FALSE.⌋

Note:
Compiler vendors that provide a boolean data type that cannot be disabled have to change their compiler (i.e. make it ANSI C compliant).

Example: API returns boolean value

```
/* File Eep_21_LDExt.h: */
...
/* this automatically includes Platform_Types.h: */

#include "Std_Types.h"
...
boolean Eep_21_LDExt_Busy(void) {...}
...
```

```
/* File: calling module */
...
if (Eep_21_LDExt_Busy() == FALSE) {...}
...
```

7.1.19 Distributed execution on multi-partitioned systems

The AUTOSAR architecture supports the execution of *BSW modules* functionality on multiple partitions, possibly running on different cores. If a module provides services on multiple partitions, then either

- the *RTE* transports the service call to the partition where the *BSW module* entity that shall execute the call is located, or
- the *BSW module* entity receives the call on the partition where it has been called and handles its execution autonomously (new in Release 4.1). That means, it can execute the call on the same partition, forward it to another partition or do a combination of both - depending on the implementation strategy of the BSW vendor.

[SWS_BSW_00190] Same API on each partition

Upstream requirements: SRS_BSW_00312

⌈If a *BSW module* entity shall be accessible from multiple partitions (e.g. multiple cores), then it shall provide the same API on each partition where the module entity shall be accessible.⌋

[SWS_BSW_00191] Multi-core safety

Upstream requirements: SRS_BSW_00312, SRS_BSW_00459

⌈If a *BSW module* entity shall be executable on multiple partitions (e.g. multiple cores), then the whole module entity code shall be *"concurrency safe"*.⌋

Note:

"Concurrency safe" refers to the overall design of the *BSW module* entity that shall be executable in multiple partitions on different cores in parallel. If, for example, the module code in different partitions accesses the same data, then the shared data shall be protected by exclusive areas.

[SWS_BSW_00192] Reentrant function code

Upstream requirements: SRS_BSW_00312

⌈If a *BSW module* entity is provided to *SWCs* and it shall be executable on multiple partitions (e.g. multiple cores), then the module entity's function code shall be implemented according to the level *"concurrency safe"*.⌋

This allows the usage of the same entry point in the code for a module function called from different partitions. The partition specific handling of the module function shall then be implemented by partition dependent branching within the module.

7.1.20 Libraries

7.1.20.1 Background

Libraries are different to *BSW modules* in several aspects although they are both specified in a software specification (*SWS*). The most prominent differences are described in the table below:

	Library	**BSW module**
Initialization	Has no `Init()` (nor a `DeInit()`) function. Functions can be used directly.	Requires initialization, may support a de-initialization. Calling functions of an uninitialized module may cause a `E_UNINIT` error.
Who uses it?	*SWC*, *BSW modules* and other libraries can directly call library functions	*BSW modules* can directly call module functions. *SWCs* can only access functionality via Ports.
Can ... call libraries?	Yes	Yes
Can ... call BSW modules?	No. Calls to *BSW modules* from libraries are forbidden. This means that there is no error reporting to *Det* or *Dem* possible. Also calls to Os are not possible.	Yes
Has configuration?	Mostly no	Mostly yes
Reentrancy?	All functions are reentrant. If some sort of a *state* is required it must be provided by the user (e.g. as a *handle*). Libraries are stateless.	Functions may or may not be reentrant
Asynchronous/Synchronous?	Only synchronous functions are possible. Asynchronous behavior would require some state handling.	Both types are possible
Error handling	Mostly no error handling. Libraries assume a responsible caller. No calls to *Det* or *Dem*.	Via return values and reporting to *Det*/*Dem*

To easily distinguish between libraries and BSW modules, the (long) title of a library includes the term *Library*.

7.1.20.2 Configuration

A library can have a configuration, but this is discouraged. In AUTOSAR most libraries do not have a configuration because a library has a global scope and can be used by

any *SWC* and *BSW module* directly. Furthermore there is no `Init()` function which could take the configuration pointer (no variations possible).

When a configuration is needed special care has to be taken that there are no dependencies to other *BSW module* configuration is introduced or the reentrancy is violated. The type of configuration must not depend on some sort of initialization. Therefore only pre-compile configuration options are possible.

Example for a configuration of a library could be that the library offers different optimizations, e.g. optimization for execution speed (e.g. realized by a table lookup) or optimization for size. Be aware that in this example the selected optimization is then globally valid for all users.

7.1.20.3 Requirements

[SWS_BSW_00259] Shared library code ⌈A library shall be implemented in a way that the code can be shared among callers in different memory partitions. A direct call from *SWC* and *BSW modules* shall be possible.⌋

[SWS_BSW_00260] Call context of library users ⌈A library implementation does not offer an `Init()` function and shall be usable as soon as the C startup is finished. This means that library functions can be called from all C environments and contexts.⌋

[SWS_BSW_00261] Calls from library code ⌈A Library function shall only call other library functions but not call any *BSW module* functions.⌋

[SWS_BSW_00267] Only direct calls to libraries ⌈The API of *Libraries* does only support direct calls. A Library does never offer a port interface. All callers - including calls from *SWCs* - are only using direct calls.⌋

[SWS_BSW_00262] Only limited error handling ⌈All Library functions shall avoid handling user faults and values outside specified range. Providing input values which are outside the specified range and/or not handled in the specification result in undefined behavior.⌋

The following example shows a wrong usage of a library function with a parameter outside the specified range that leads to memory corruption:

```
1 /* There is an array with 5 elements */
2 float32 Array[5];
3
4 /* This call sorts 7 elements at the memory address of the array */
5 float32 Median = Mfl_MedianSort_f32_f32(Array, 7);
6
7 /* Which means that it corrupts 8 bytes (2 float32 elements) after the
      array, which belong to entirely different variables */
```

A mitigation measure has to be implemented in the user code itself, e.g. by using a pre-defined or constant array size:

```
/* There is an array with 5 elements */
#define ARRAY_SIZE 5
float32 Array[ARRAY_SIZE];

/* This call sorts ARRAY_SIZE elements at the memory address of the
   array */
float32 Median = Mfl_MedianSort_f32_f32(Array, ARRAY_SIZE);

/* Which means that it correctly stays within the memory boundaries of
   the array */
```

7.1.21 Asynchronous copy approach (DRAFT)

7.1.21.1 Background

In some places of the communication stack it is beneficial to copy data asynchronously by offloading the copy process to a DMA-engine. Most likely the data path of the communication stack is associated with `Pdu`s that are collected in a virtual ECUC module (see [6]). Figure 7.1 shows the priniciple of the module layering and the referenced `Pdu`s.

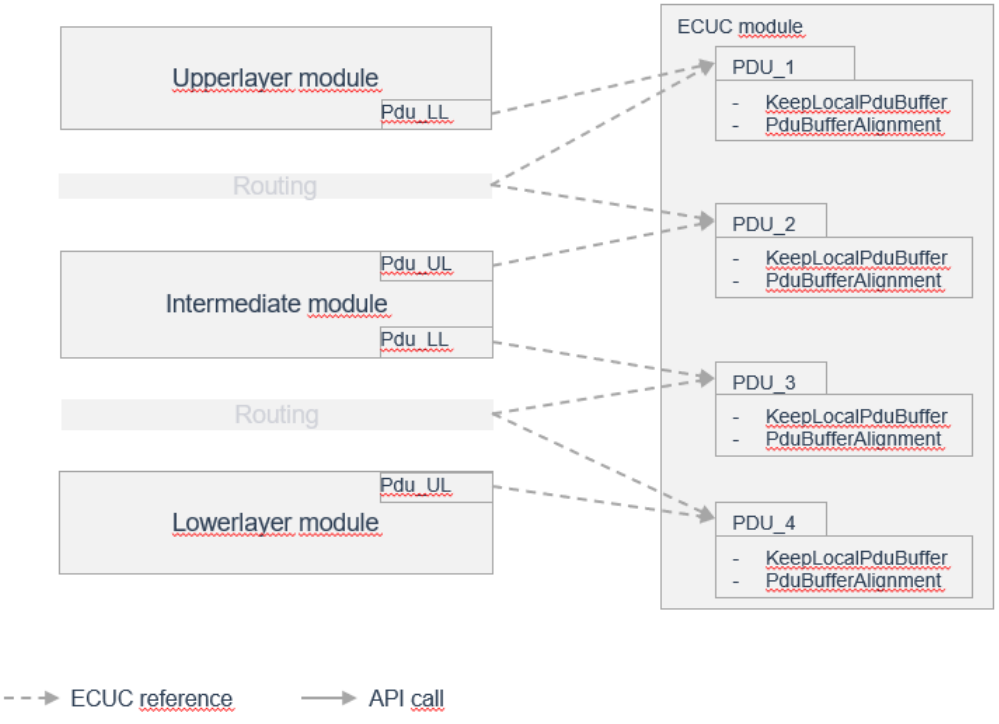


Figure 7.1: Priniciple of module layering and Pdu configuration

Each module may provide buffers. Buffers which are provided are called "source buffers". Buffers which are allocated locally to consume data are called "destination buffers". A module with consumes the source buffer may copy the data to its local buffer. Copying could be performed synchronously in context of an transmit rerquest or reception indication, or asynchronously by offloading the copy process to a DMA-

engine. Synchronous copy is excuted by the CPU and called "CPU-copy". Asynchronous copy could be executed by a DMA-engine and called "DMA-copy", and if the start memory addresses of the source buffer is misaligned and/or bytes are left-over at end which needs to be copied, then additionally CPU-copy needs to be executed. A *BSW module* decides based on the configuration if synchronous or asynchronous copy could be used:

- `EcucDmaThreshold` defines a threshold to decide for which data length a DMA-copy would be beneficial
- `KeepLocalPduBuffer` defines if the source buffer and any other buffer in between which is associated with this `Pdu` would be keept available after a transmission request or reception indication returns, or if the the source buffer is released with the return of the function call

A module which is requested to transmit data or indicated to receive data would request an asynchronuous copy process, if the provided data length exceeds the configured `EcucDmaThreshold` and the associcated `Pdu` has `KeepLocalPduBuffer` set to `TRUE`. The source buffer needs to be released after the asynchronous copy is finalized (successful or unsuccessful). On the transmisson path the destination module calls transmission acknowledgment (see Figure 7.2) to release the source buffer.

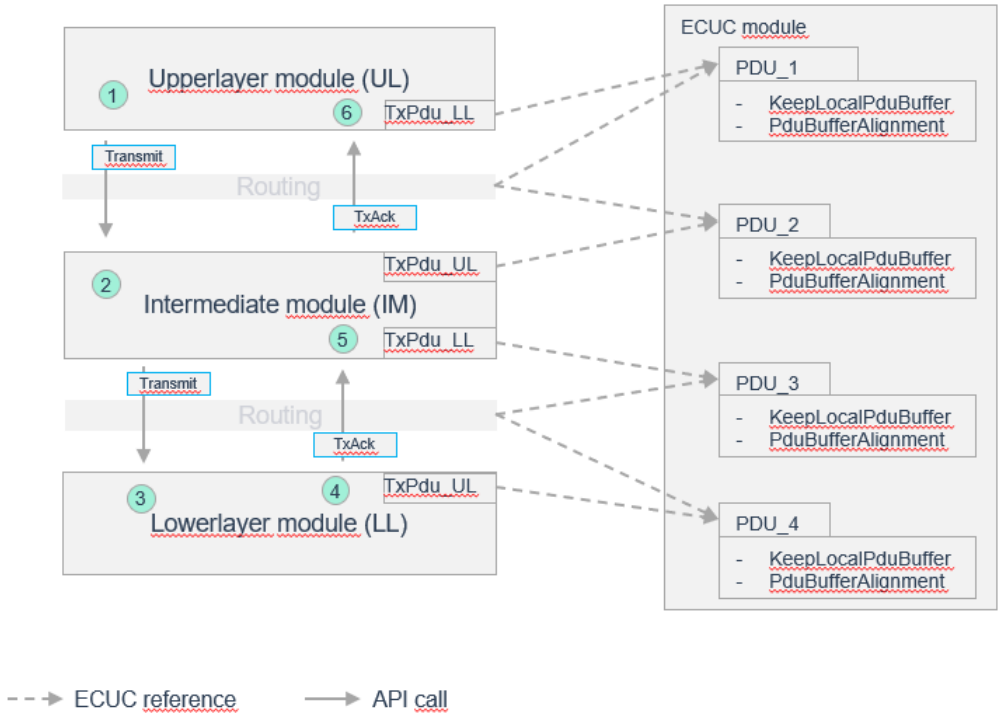


Figure 7.2: Priniciple of asynchronous copying within a transmission request

A upperlayer module (1) add data into its source buffer and calls a transmission request. The transmission request is forwarded by the intermediate module (2) to the lowerlayer module (3). The lowerlayer module may copy the data of the source buffer via an asynchronous copy process to its local buffer (destination buffer). After copying is finalized and data of the destination buffer has been transmitted, a transmission

acknowledgment is indicated (4). The intermediate module forward the transmission acknowledgment (5) and the upperlayer module release the source buffer with the arrival of the transmission acknowledgment (6).

On the reception path the destination module calls release reception buffer api (see Figure 7.3) to release the source buffer.

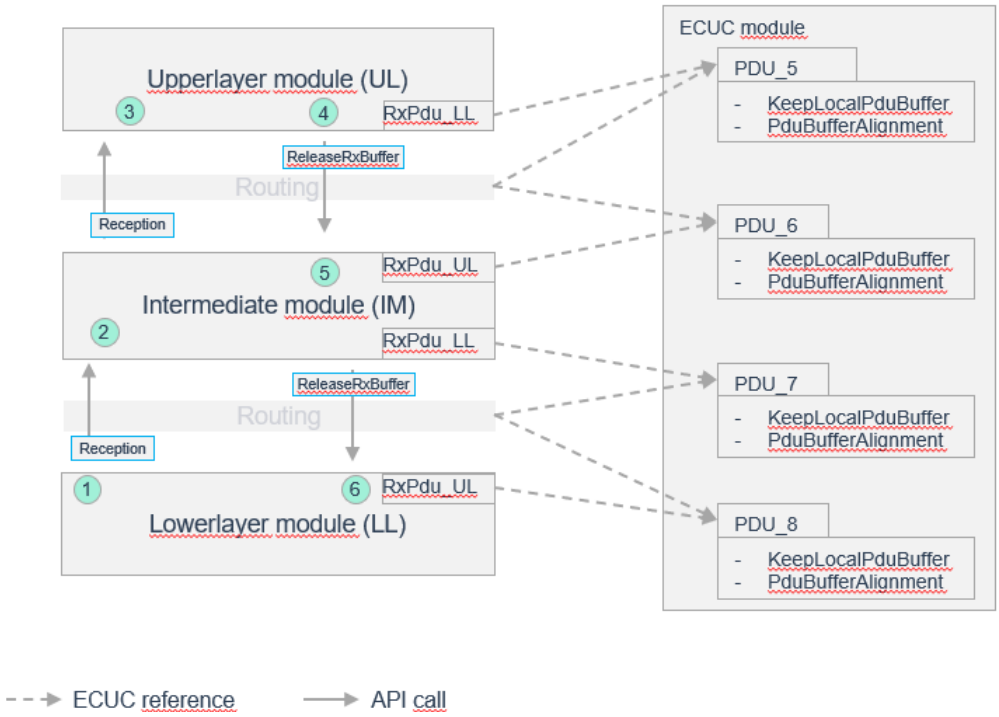


Figure 7.3: Priniciple of asynchronous copying within a reception indication

A lowerlayer module (1) received data at its source buffer and calls a reception indication. The reception indication is forwarded by the intermediate module (2) to the upperlayer (3). The upperlayer may copy the data of the source buffer via an asynchronous copy process to its local buffer (destination buffer). After copying is finalized, a releae reception buffer is indicated (4). The intermediate module forwards the release reception buffer indication (5) and the upperlayer module releases the source buffer with the arrival of the release reception buffer call (6).

DMA-copy need to respect the required buffer aligment to ensure stable performance (e.g. throughput of data). The buffer alignment is configured on driver level and propagated at configuration time to the *BSW modules*. Therefore a `Pdu` has a configuration parameter `PduBufferAlignment`. When ever a source buffer is allocated and asynchronous copy is offered by the source *BSW module*, then the memory start address of the source buffer need to respect the configured buffer alignment. A *BSW module* which allocates the destination buffer needs to respect the configured buffer aligment as well, and in addition, it needs to check if the given source memory start address is aligned according the configured buffer alignment. For example, at the reception path it could happen that the provided source memory start address is misaligned.

When ever a message frame is received, then this data is added to the lowerlayer source buffer. Each layer may inspect the parts of the data and push the source buffer pointer to the boundary, where the next following upper *BSW module* is responsible for (see Figure 7.4).

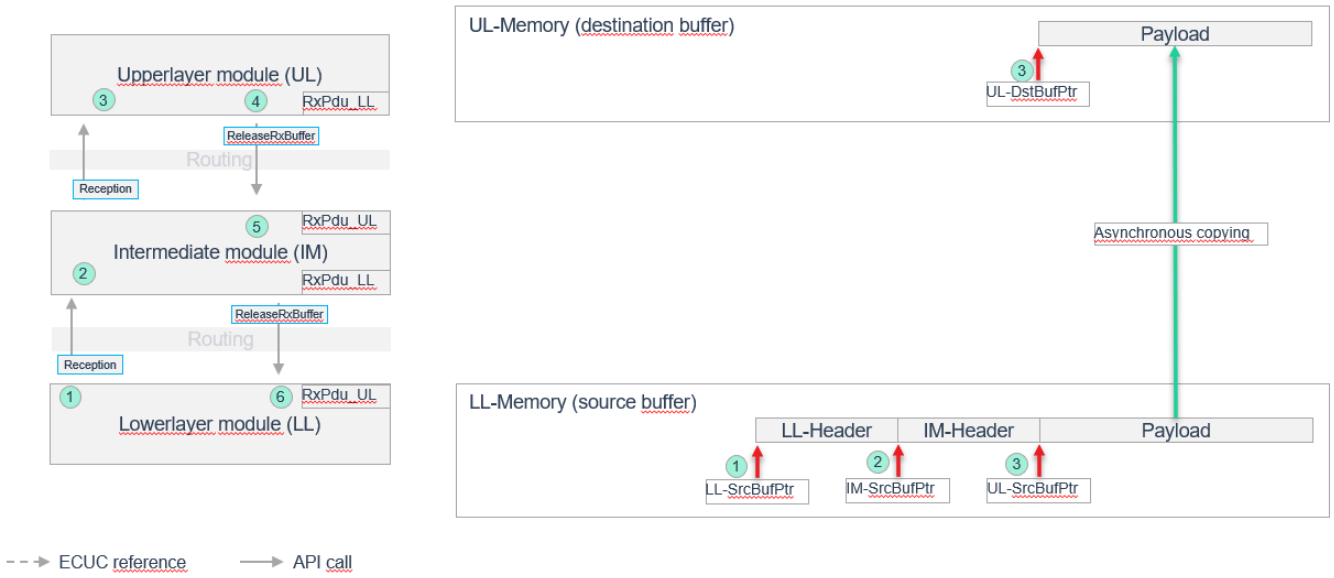


Figure 7.4: Priniciple of asynchronous copying with focus on the source buffer alignment

A *BSW module* which provides a destination buffer needs to allocate a buffer length that gives the flexibility to start a copying process at an misaligned memory address. When a source buffer with a misalinged memory start address is detected by the destination *BSW module*, then the *BSW module* need to consider to start on a misaligned memory start address of the destination pointer as well, and copy data with CPU-copy to next aligned memory address of the source buffer. From this aligned memory address the destination module need to calculate the length, which could be copied via DMA-copy. Therefore the destinantion module need to consider the maximum length that do not exceed the source buffer length and respect the configured buffer alignment. Remaing bytes at the end of the source buffer may need to be copied via CPU-copy to the destination buffer (see Figure 7.5)

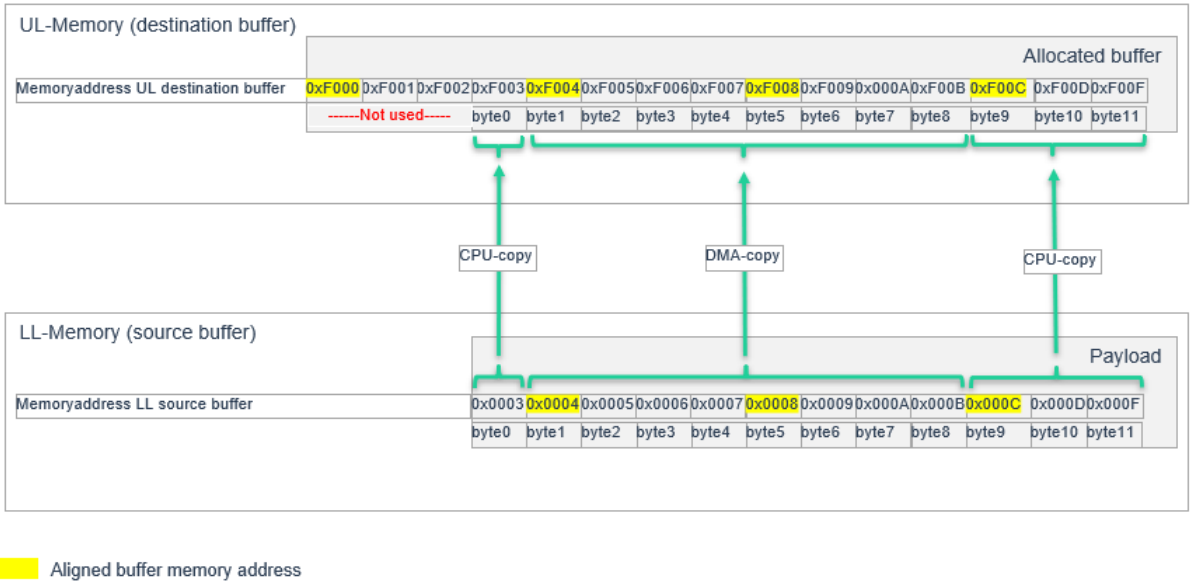


Figure 7.5: Details on handling misaligned source buffer memory address. Assumption: 4 byte buffer alignment, 12 byte source buffer length

7.1.21.2 Requirements

[SWS_BSW_00265] Consider configured buffer alignment for allocation of a destination buffer

Status: DRAFT
Upstream requirements: SRS_BSW_00426

⌈If a *BSW module* need to copy data to its local buffer (destination buffer) and the *BSW module* which provides the data (source buffer) offer the possibility to perform an asynchronous copying (e.g. a `Pdu` of the virtual ECUC module (see [6]) associated with the source buffer has `KeepLocalPduBuffer` is set to `TRUE`), then the length of the destination buffer shall be calculated with the following equation:

$$lenght_{\text{destination buffer}}[bytes] = length_{\text{data}}[bytes] + (alignment[bytes] - 1) \quad (7.1)$$

⌋

Note: A configured alignment could be derived for example from an `Pdu` (see `PduBufferAlignment` at [6]).

[SWS_BSW_00266] Support for asynchronous copy operations

Status: DRAFT
Upstream requirements: SRS_BSW_00426

⌈If a *BSW module* supports asynchronous copying (e.g. DMA), it shall be able to handle buffer alignment mismatches between source and destination buffers by ensuring proper destination buffer allocation, address alignment, and, where necessary, falling back to synchronous copying for unaligned portions of the data.⌋

7.2 Error Handling

Particular errors are specified in Chapter 7 of the respective *BSW Module* specifications.

The following section forms the foundation for this. Above all, it specifies a classification scheme consisting of five error types that may occur in *BSW modules* during different life cycles.

7.2.1 Classification

[SWS_BSW_00144] Error classification

Upstream requirements: SRS_BSW_00385

⌈All errors, which may be detected and/or reported by a *BSW Module*, are classified by the following types:

- development errors [SRS_BSW_00337]
- runtime errors [1, SRS_BSW_00452]
- production errors [1, SRS_BSW_00458]

⌋

7.2.2 Development errors

7.2.2.1 Synopsis

*Development errors*s are mainly specified as software bugs which occur during the software development process, cf. [SRS_BSW_00337] for the detailed specification.

For instance, the attempt to use uninitialized software is a typical *development error*.

Development errors are reported to the *BSW module Det (Default Error Tracer)* through the interface `Det_ReportError`, which also reflects the event-oriented character of this type. *Development errors* eventually happen and corresponding error monitors will immediately signal their occurrence.

Although the specification document of the module *Det* does not specify any particular behavior or implementation, [SRS_BSW_00337] requires that *development errors* behave like assertions. Their appearance will abort the normal control flow of execution by halting or resetting of the entire ECU.

In addition to the *Development Errors* specified by AUTOSAR SWS documents, implementations of *BSW modules* may define additional *Development Errors* to suit user needs, to cover unique technological aspects of the implementation, etc. In this case, it would be beneficial to provide documentation to the user describing these additional *Development Errors* and how they are utilized in the implementation.

[SWS_BSW_00255] Order of development error checks

Upstream requirements: SRS_BSW_00337

⌈There is no defined order of *development error* checks, except that a check for uninitialized module (`<MIP>_E_UNINIT`) shall be performed first.⌋

7.2.2.2 Documentation

The SWS shall list the development errors in its chapter 7 in accordance with the classification of [SRS_BSW_00337].

[SWS_BSW_00201] *Development error* type

Upstream requirements: SRS_BSW_00337

⌈*Development error* values are of type `uint8`.⌋

7.2.2.3 Configuration

[SWS_BSW_00203] API parameter checking

Upstream requirements: SRS_BSW_00323

⌈If the `<Ma>DevErrorDetect` switch is enabled API parameter checking is enabled. The detailed description of the detected errors can be found in chapter 7.2 and chapter 8 of the respective module SWS.⌋

[SWS_BSW_00042] Detection of *Development errors*

Upstream requirements: SRS_BSW_00337

⌈The detection and reporting of *Development errors* shall be performed only if the configuration switch `<Ma>DevErrorDetect` is enabled. The default value of the configuration switch shall be that detection and reporting is disabled.⌋

The detection of *development errors* is configurable. It enables extended debugging capabilities for the according *BSW Module*.

Example: The EEPROM driver provides internal checking of API parameters which is only activated for the first software integration test (*"development build"*) and disabled afterwards (*"deployment build"*).

Example:

The implementation code is generated automatically by the supporting tool chain considering the configuration parameter for the detection of *Development errors*. If the detection is not configured, the generated code does not contain error detection and reporting implementation.

Example:

The implementation code contains compiler switches, which implement the configuration of error detection:

```
/* File: Nm_Cfg.h */
/* Pre-compile configuration parameters for Network Manager */
...
/* NM_DEV_ERROR_DETECT */
/* To activate (STD_ON) or deactivate (STD_OFF) detection of */
/* development errors. */
/* Satisfies BSW_SWS_042. */
#define NM_DEV_ERROR_DETECT STD_ON
...
```

```
/* File: Nm.c */
/* Network Manager implementation */
...
#include "Nm_Cfg.h"
...
#if ( NM_DEV_ERROR_DETECT == STD_ON )
...
... /* development errors to be detected */
...
#endif /* NM_DEV_ERROR_DETECT */
```

Note that for switching this configuration through compiler switches the standard types `STD_ON` and `STD_OFF` shall be used [SWS_BSW_00029].

The configuration parameter for detection of *Development errors* is listed in the Chapter 10 of the respective *BSW Module* specification.

If the detection of *Development errors* is active, then API parameter checking is enabled [SWS_BSW_00049]. The detailed description of the detected errors can be found in chapter 7 and chapter 8 of the according *BSW Module* specification.

7.2.2.4 Reporting

If the detection of *Development errors* is configured (see [SWS_BSW_00042]) than any detected error shall be reported:

[SWS_BSW_00045] Report detected *Development errors* to *Det*

Upstream requirements: SRS_BSW_00337

⌈The *BSW Module* shall report detected *Development errors* to the *Default error tracer (Det)* using the service `Det_ReportError` with its assigned module identifier (see Appendix "List of Basic Software Modules") to identify itself.⌋

Note that the reported development error values must be of type `uint8`, in order to comply with the signature of `Det_ReportError`.

See chapter 7.2.2 - ***Development errors*** for more information about activation and deactivation of *Development error* detection. See the Specification of *Det* [14] for more information about the service `Det_ReportError`.

[SWS_BSW_00243]

Upstream requirements: SRS_BSW_00337, SRS_BSW_00487

⌈*BSW Modules* with enabled *development error* detection shall raise the development error `<MIP>_E_UNINIT` when any function apart from the `<Mip>_Init` and the *schedule functions* (see [SWS_BSW_00037]) are called for an un-initialized *BSW Module*.⌋

[SWS_BSW_00264] Calling initialization functions multiple time

Upstream requirements: SRS_BSW_00337, SRS_BSW_00498

⌈*BSW Modules* with enabled *development error* detection shall raise the development error `<MIP>_E_ALREADY_INITIALIZED` when the initialization function of the module is called while the module is already initialized.⌋

7.2.3 Runtime errors

7.2.3.1 Synopsis

Runtime errors occur during runtime of the ECU, and are caused by faults stemming from software or hardware issues. *Runtime errors* do not necessarily affect the overall system behavior.

As examples, *runtime errors* can be caused by queue overflow in the memory stack or an API is called at the wrong time. Also, *runtime errors* could occur if thermal noise or particle radiation influences the functionality of hardware and so the functionality of the software connected with it. This could also mean that *runtime errors*s may heal, because the cause for the fault may disappear, again. For instance, a CAN controller HW could go off-line due to a bit-flip in its control registers, induced by particle radiation.

Like *development errors*, *runtime errors* are reported to the *BSW module Det*, in this particular case through the interface `Det_ReportRuntimeError`. Just as *development errors*, *runtime errors* also eventually happen and cause the corresponding error monitors to signal their occurrence immediately.

Unlike *development errors* however, *runtime errors* shall not cause assertions, i.e., the control flow of execution will continue. Instead of that, an occurrence of a *runtime error* triggers the execution of a corresponding error handler. This error handler may be implemented as callout within the *Det* by an integrator of a particular ECU and may only include the storage of the corresponding error event to a memory, a call to the module *Dem* or the execution of short and reasonable actions.

The *Det* module provides an optional callout interface to handle *runtime errors*. If it is configured, the service `Det_ReportRuntimeError` shall call this callout function. Independent from any particular implementation, the service `Det_ReportRuntimeError` always returns `E_OK` to its caller.

Monitors dedicated to detect *runtime errors* may stay in the deployment build (production code).

7.2.3.2 Documentation

The SWS shall list the runtime errors in its chapter 7 in accordance with the classification of [1, SRS_BSW_00452].

[SWS_BSW_00219] *Runtime error* type

Upstream requirements: SRS_BSW_00452

⌈*Runtime error* values are of type `uint8`.⌋

7.2.3.3 Configuration

Runtime errors can not be switched off (like *development errors*) via a configuration parameter.

If the *Det* implements the handling of *runtime errors* by a callout function, then the particular callout function name of the *Det* must be configured by `DetReportRuntimeErrorCallout` [14, ECUC_Det_00010].

7.2.3.4 Reporting

Any detected *runtime error* shall be reported:

[SWS_BSW_00222] Report detected *Runtime errors* to *Det*

Upstream requirements: SRS_BSW_00385, SRS_BSW_00452

⌈The *BSW Module* shall report detected *runtime errors* to the *Default error tracer (Det)* using the service `Det_ReportRuntimeErrors`.⌋

Note that the reported *runtime error* values must be of type `uint8`, in order to comply with the signature of `Det_ReportRuntimeError`.

See chapter 7.2.3 ***Runtime errors*** activation and deactivation of *Development error* detection. See the Specification of *Det* [14] for more information about the service `Det_ReportRuntimeError`.

7.2.4 Production errors

7.2.4.1 Synopsis

According to [1, SRS_BSW_00458] *production errors* are caused by any hardware problems, e.g., aging, deterioration, total hardware failure, bad production quality, in-

correct assembly, etc. These hardware problems qualify for being production errors, if at least one of the following criteria is met (cf. [1, SRS_BSW_00458]):

- The error leads to an increase of emissions and must be detected to fulfill applicable regulations.
- The error limits the capability of any other OBD relevant diagnostic monitor.
- The error requests limp-home reactions, e.g., to prevent further damage to the hardware or customer perceivable properties.
- The garage shall be pointed to the failed component for repair actions.

In addition, [1, SRS_BSW_00458] and [1, SRS_BSW_00472] require to avoid duplicate *production errors* which have the same root cause as failure. This means in first place that the specification of particular *production errors* need some wider scope than only the one of a specific *BSW module* (see DemComponent in Dem [15]).

A particular *production error* is reported to the module *Dem* and may utilize all available features of it. In general, any 'fail' of a corresponding error monitor will lead to an entry into the primary event memory, a 'pass' may revoke this entry.

It is generally possible to combine distinct options of the *Dem* for a single *production error*. Thus, a particular *production error* may lead to an entry in the primary event memory and may trigger a dedicated callout routine that utilizes its states for deduced actions, at the same time.

7.2.4.2 Documentation

[SWS_BSW_00204] Documentation of *production errors*

Upstream requirements: SRS_BSW_00458

⌈For each *production error*, appropriate documentation shall be provided according to the AUTOSAR SWS template.⌋

7.2.4.3 Configuration

[SWS_BSW_00205] Detection of *production errors*

Upstream requirements: SRS_BSW_00385, SRS_BSW_00458

⌈The detection of *production code errors* cannot be switched off, unless the Module SWS describes configuration parameters or other conditions, which define the activation of certain *production errors*.⌋

7.2.4.4 Reporting

Event IDs of *production errors* are provided as symbolic name values by *Dem* through `Dem.h`.

The `EventId` symbols of *production errors* are the short name of the `ServiceNeeds` of the *BSW module* (through the *Dem* ECUC) prefixed with `DemConf_DemEventParameter_`.

See **TPS_ECUC_02108** from *AUTOSAR Specification of ECU Configuration* [6].

[SWS_BSW_00143] Values for Event IDs of *production errors* are imported

Upstream requirements: SRS_BSW_00409

⌈Values for *Event IDs* of *production errors* are assigned externally by the configuration of the *Dem* module.⌋

For reporting *production errors*, the *Dem* interface `Dem_SetEventStatus` is used:

[SWS_BSW_00046] Report *production errors* to *Dem*

Upstream requirements: SRS_BSW_00339

⌈The *BSW Module* shall report all detected *production errors* to the *Diagnostic Event Manager (Dem)* using the service `Dem_SetEventStatus` if this specific *production error* has been configured for this *BSW Module*.⌋

Note that the configuration of production errors is optional in the ECU Configuration of the BSW Modules.

[SWS_BSW_00066] Report EventStatus to *Dem*

Upstream requirements: SRS_BSW_00339

⌈For reporting an *production error* state the following BSW specific interface of *Dem* shall be called:

```
Std_ReturnType Dem_SetEventStatus (
    Dem_EventIdType EventId,
    Dem_EventStatusType EventStatus
)
```

If an error event occurred `EventStatus` shall be equal to: '`DEM_EVENT_STATUS_FAILED`'.

If an error event is not detected with sufficient precision and requires maturing by pre-debouncing `EventStatus` shall be equal to: '`DEM_EVENT_STATUS_PREFAILED`'.

If the *BSW modules* has explicitly detected that the error is not present `EventStatus` shall be equal to: '`DEM_EVENT_STATUS_PASSED`'.

If a failure free detection is not possible with sufficient precision and requires further maturing by pre-debouncing `EventStatus` shall be equal to: '`DEM_EVENT_STATUS_PREPASSED`'.

If a check is not possible (e.g., requires specific operating mode), no result shall be reported.⌋

Note:
The return value of `Dem_SetEventStatus` shall be ignored by the BSW modules.

The error state information could be reported either by a state change or when the state is checked (event or cyclic) depending upon the configuration of the error event. Checks are not required to be cyclic.

Pre-de-bouncing is handled inside the *Diagnostic event manager* using AUTOSAR pre-defined generic signal de-bouncing algorithms.

Note:
The callback service `<Mip>_DemInitMonitorFor<EventName>` is principally specified by the specification [SWS_Dem_00256] within Section 8.4.3.1.1 of the specification document for the module *Diagnostic Event Manager (Dem)* [15]. This document only specifies extensions which matter for the correct functionality of error monitors.

[SWS_BSW_00206]

Upstream requirements: SRS_BSW_00470

⌈Only event-based error monitors shall implement the callback service `<Mip>_DemInitMonitorFor<EventName>`.⌋

Note:
The *BSW module Dem* calls an implemented callback service `<Mip>_DemInitMonitorFor<EventName>` to trigger the re-initialization of an event-based error monitor depending on the `EnableConditions` or `ControlDTCSettings`. The re-initialization reason is passed by the parameter `InitMonitorReason`.

[SWS_BSW_00207]

Upstream requirements: SRS_BSW_00339

⌈On each trigger of the callback service `<Mip>_DemInitMonitorFor<EventName>`, the particular *BSW module* shall re-initialize the monitor functionality and report a new error status to the *BSW module Dem* immediately, if the error status could be evaluated anytime, otherwise at the next opportunity.⌋

[SWS_BSW_00208]

Upstream requirements: SRS_BSW_00339, SRS_BSW_00334

⌈If a particular *BSW module* implements a callback service `<Mip>_DemInitMonitorFor<EventName>`, then the BSWMD shall specify a corresponding `ServiceNeeds`.⌋

7.2.4.5 Example use case: Error is detected and notified

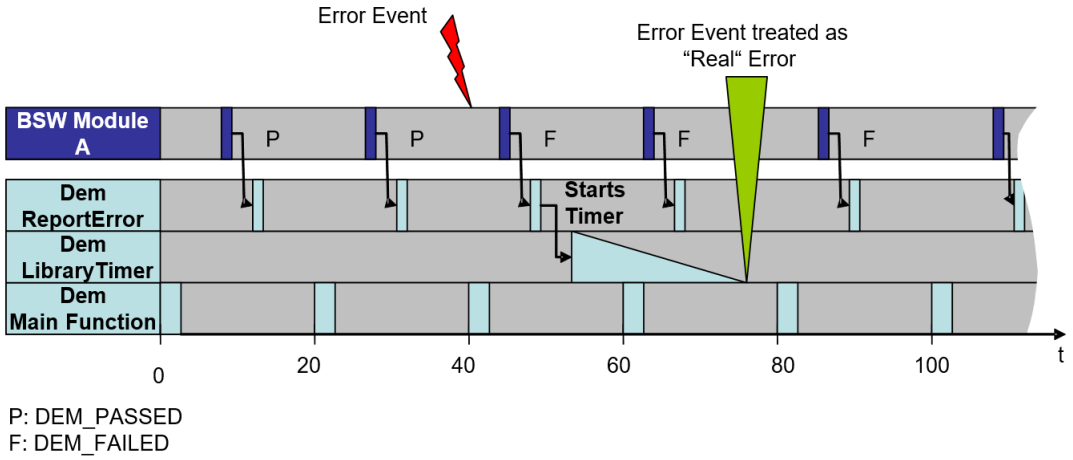


Figure 7.6: Example of error reporting to Diagnostic event manager

The timer function shall be provided (in this example) in the pre-de-bouncing library of the *Diagnostic event manager*.

7.2.5 Specific topics

7.2.5.1 Implementation specific errors

[SWS_BSW_00073] Implementation specific errors

Upstream requirements: SRS_BSW_00337

⌈If the *BSW Module* implementation defines additional errors, then these shall be described in the *BSW module* documentation. The error classification table shall be extended by implementation specific errors.⌋

7.2.5.2 Handling of Symbolic Name Values

[SWS_BSW_00200] Symbolic Name values

Upstream requirements: SRS_BSW_00302

⌈Symbolic Name Values shall be imported through the header of the *BSW module* that provides the value.⌋

Symbolic Name Values in the implementation are using the short name of the Container in the ECUC prefixed with `<ModuleAbbreviation>Conf_` (of the producing module) and the short name of the `EcucParamConfContainerDef` container [TPS_ECUC_02108].

Example: For production errors, which are provided by the *Dem*, and are configured as `DemEventParameter` within the ECUC of the Dem, the `#define` provided through `Dem.h` is `DemConf_DemEventParameter_<short-name>`.

The following two code integration examples show the utilization of a production code event ID (14) and its symbol (`DemConf_DemEventParameter_EEP_21_LDEXT_E_COM_FAILURE`) for the module *Eep*:

- Example for **source code integration**:

```
/* File: Dem_Cfg.h */
...
/* DEM specifies the production code error ID: */
#define DemConf_DemEventParameter_EEP_21_LDEXT_E_COM_FAILURE ((
    Dem_EventIdType) 14u)
...
```

```
/* File: Eep_21_LDExt.c */
#include "Dem.h"
...
(void)Dem_SetEventStatus (
    DemConf_DemEventParameter_EEP_21_LDEXT_E_COM_FAILURE,
    DEM_EVENT_STATUS_PREFAILED);
```

- Example for **object code integration**:

```
/* File: Dem_Cfg.h */
...
/* DEM specifies the production code error ID: */
#define DemConf_DemEventParameter_EEP_21_LDEXT_E_COM_FAILURE ((
    Dem_EventIdType) 14u)
```

```
/* File: Eep_21_LDExt_Lcfg.c
Link-time configuration source
This file needs to be compiled and linked with the
object code delivery: */
#include "Dem.h"
#include "Eep_21_LDExt_Lcfg.h"
...
const Dem_EventIdType Eep_21_LDExt_E_Com_Failure =
    DemConf_DemEventParameter_EEP_21_LDEXT_E_COM_FAILURE;
...
```

```
/* File: Eep_21_LDExt_Lcfg.h
This file needs to be compiled and linked with the
object code delivery: */
...
extern const Dem_EventIdType Eep_21_LDExt_E_Com_Failure;
...
```

```
/* File: Eep_21_LDExt.c
This file is delivered as object file. */
#include "Dem.h"
#include "Eep_21_LDExt_Lcfg.h"
...
(void)Dem_SetEventStatus ( Eep_21_LDExt_E_Com_Failure,
    DEM_EVENT_STATUS_PREFAILED);
```

7.2.5.3 Handling of corrupted NV data

Sometimes BSW modules require a storage of their data in NV memory so that it is still available for future use, e.g. after a restart of the system. In classic AUTOSAR the NvM offers this functionality based on so called NV blocks. So BSW modules can use the NvM to store their data into the NV memory. As a precondition the NvM requires a related configuration of these blocks. More information can be found in [16, Specification of NVRAM Manager] and [17, NV Data Handling Guideline].

The typical use cases for the modules can be divided in:

- **Last state**: The modules want to store some state which is used during the next start of the system.
- **Cache**: The module wants to store data which it had collected before to allow a faster startup. This could include some data the module has recently learned.
- **Initialization data**: The module was dynamically reconfigured and needs to preserve the new configuration data.
- other cases and/or a combination from the above cases

What happens when the access to the NV data is not possible, e.g. because the related memory has problems? As the issue is specific to the module which requires the data, no general answer can be given. The following aspects should be considered:

- Does the problem occur during a read or write or both operations?
- Does the problem occur during the start of the module? Or does it happen during the "normal use"?

The following hints may help to resolve the situation. If a module requires a standardized behavior, then this needs to be specified in the related SWS of the module. In general modules could do the following:

- If the problem is happening during startup and the missing NV data is essential, it might be good to stop the system at this point and to raise a production error.
- If the NV data is not essential, the module might fall back to some default values. This could be realized by initial values stored in ROM (for initialization data, e.g. "Last state" case from above). In the "Cache" case it might be useful to start with an empty cache.
- The NvM can be configured to provide an init block when reading is not possible and/or redundant blocks. With redundant blocks the likelihood to read valid data increases. When no valid data can be read, an init block replaces the values of the NV memory.
- A retry of the failed read or write operation might be useful.

The handling of such error situations depends on the module and the usage of the NV data. If the related module specification (SWS) does not specify the handling for non-essential errors, the following behavior is expected:

[SWS_BSW_00263] Handling of non essential NV data errors

Upstream requirements: SRS_BSW_00497

⌈If a module using the NvM encounters access problems to non essential NV data, then the module shall handle the situation internally and report a runtime error to Det. The name of the runtime error should follow the pattern <MIP>_E_NVM_ACCESS_FAILED.⌋

For essential errors no general handling can be prescribed in this document. The module specification (SWS) should contain the intended behavior.

7.3 Security events

7.3.1 Synopsis

Onboard *security events* are reported by security related *BSW modules* or *SWCs*. They shall be reported to the *Intrusion Detection System Manager (IdsM)*.

Events are regarded as *security events*, if at least one of the following criteria are met:

- The event indicates a successful execution of a security related function which shall be captured for later analysis. An example of such an event is the successful installation of a certificate.
- The event indicates a failed security check which shall be recorded for later analysis. An example for such an event is the failed MAC verification of a secured PDU.

7.3.2 Documentation

[SWS_BSW_00244] Documentation of *security events*

Upstream requirements: SRS_BSW_00488

⌈For each *security event*, appropriate documentation shall be provided according to the AUTOSAR SWS template.⌋

7.3.3 Configuration

[SWS_BSW_00250] Enabling and disabling reporting of *security events*

Upstream requirements: SRS_BSW_00489

⌈The *BSW module* shall provide a configuration switch to enable reporting of *security events* from the *BSW module* to the *IdsM*. The switch shall be located in the General container of the module.

Shortname: <ModuleName>EnableSecurityEventReporting

Longname: Enable Security Event Reporting

The default setting of the switch shall be Off.⌋

7.3.4 Reporting

Event IDs of *security events* are provided as symbolic name values by *IdsM* through `IdsM.h`.

The EventId symbols of *security events* are the short name of the `SwcServiceDependency` of the *BSW module* (through the *IdsM* ECUC) prefixed with `IdsMConf_IdsMEvent_`.

See **TPS_ECUC_02108** (*AUTOSAR Specification of ECU Configuration* [6] **"2.4.5.2 Representation of Symbolic Names"**).

[SWS_BSW_00251] Values for *Event IDs* of *security events* are imported

Upstream requirements: SRS_BSW_00493

⌈Values for *Event IDs* of *security events* are assigned externally by the configuration of the *IdsM* module.⌋

[SWS_BSW_00258] Report *security events* to *IdsM*

Upstream requirements: SRS_BSW_00489

⌈The *BSW Module* shall report the detected security events to the *Intrusion Detection System Manager (IdsM)* using the service `IdsM_ReportSecurityEvent` if this specific *security event* has been configured for this *BSW Module*.

```
void IdsM_ReportSecurityEvent (
    IdsM_SecurityEventIdType securityEventId,
    const uint8* contextData,
    uint16 contextDataSize,
    uint16 contextDataVersion,
    uint16 count,
    const IdsM_TimestampDataType* timestamp
```

⌋

7.4 Meta Data Handling

Meta data of *PDU*s is supported by a large number of modules of the communication stack. It serves to transport information through the layers, that is in general abstracted by the layered architecture. The first supported *meta data* was the CAN ID of *PDU*s related to *CanIf*.

The *meta data* is transported by the `PduInfoType` structure via a separate pointer to a byte array alongside the length of and a pointer to the payload of the *PDU*. The

content of the *meta data* is defined by the *EcuC* description of the global *PDU* (`/EcuC/EcucConfigSet/EcucPduCollection/Pdu`), which gives types (`MetaDataItemType`), lengths (`MetaDataItemLength`) and the ordering of meta data items (`MetaDataItem`) contained in the *meta data* of a certain *PDU*.

[SWS_BSW_00239] Order and Position of *Meta Data* Items

Upstream requirements: SRS_BSW_00496

⌈The sequence and position of *meta data* items within the byte array containing the meta data is given by the configuration of the meta data items and their length (`MetaDataItemLength`) in the *EcuC*. The ordering by length (`MetaDataItemLength`) ensures that no padding is required within the *meta data* (i.e., between different *meta data* items) allowing the *meta data* items to be densely packed within the *meta data* array.⌋

A *PDU* has always an originating (producing) module, and a final (consuming) module, and possibly a number of intermediate (forwarding) modules. The layout of the meta data is fixed for a *PDU*. Therefore the originating module allocates the space for the complete *meta data* (i.e., for all *meta data* items), but each module along the chain of modules accessing the same *PDU* will only access the *meta data* items known to them.

[SWS_BSW_00240] Allocation of *Meta Data*

Upstream requirements: SRS_BSW_00496

⌈The first module that references a global *PDU* (`/EcuC/EcucConfigSet/EcucPduCollection/Pdu`) in a certain direction (the producing module) assembles the data of the *PDU*. It shall allocate space for the entire *meta data* defined for the *PDU*, even when it supports only a subset of the contained *meta data* items. Only the known subset of *meta data* items shall be initialized by the producing module.⌋

For example, *meta data* might by created by the *CanIf* as a CAN ID attached to an N-PDU. This meta data is then consumed by the *CanTp*, which creates a SOURCE_ADDRESS_16, a TARGET_ADDRESS_16, and an ADDRESS_EXTENSION_8 from the CAN ID, and attaches them to an N-SDU, which is then forwarded (untouched) by the *PduR* and consumed by the *DCM*. When (due to wrong configuration) an ETHERNET_MAC_64 was attached to the N-PDU, it would have been allocated by the *CanIf*, but neither initialized by *CanIf*, nor accessed by *CanTp*.

[SWS_BSW_00245] Byte Order of *Meta Data*

Upstream requirements: SRS_BSW_00496

⌈To be able to access *meta data* items by casting to the proper base type (according to `MetaDataItemType`), all *meta data* items shall be stored in platform byte order.⌋

[SWS_BSW_00241] Alignment of *Meta Data*

Upstream requirements: SRS_BSW_00483, SRS_BSW_00496

⌈To be able to access *meta data* items by casting to the proper base type (according to `MetaDataItemType`), the whole *meta data* array allocated by the producing module needs to be aligned according to the most stringent alignment requirements of all the contained *meta data* items.⌋

For example, the *meta data* array for *meta data* consisting of *meta data* items of type SOURCE_ADDRESS_16, ADDRESS_EXTENSION_8, and ETHERNET_MAC_64 has to be 64 bit aligned.

[SWS_BSW_00242] Access to *Meta Data*

Upstream requirements: SRS_BSW_00496

⌈Each module that references a global *PDU* including *meta data* shall only access (read and/or write) the *meta data* items that it knows. Unknown *meta data* items shall be left untouched.⌋

8 API specification

8.1 Imported types

A list with imported types and the according included header files is specified in chapter 8 of the according *BSW Module* specification. Imported type definitions are defined using the following template:

Module	***Available via***	***Imported Type***
Module name	Header which defines these types.	Name of the imported type1
		Name of the imported type2
		...
Another Module name	Header which defines this type.	Name of the imported type.
...	...	...

Table 8.1: Imported types template

8.2 Type definitions

[SWS_BSW_00146] Naming conventions for data types

Upstream requirements: SRS_BSW_00305

⌈All data types defined by the *BSW Module*, except ConfigType, shall be labelled according to the following convention:

`<Ma>_<Tn>Type`

Where `<Ma>` is the *Module abbreviation* ([SWS_BSW_00101]) and `<Tn>` is the *Type name*, which shall be written in *camel case*.⌋

Examples:

- `Eep_LengthType`
- `Dio_SignalType`
- `Nm_StateType`

Note that Basic AUTOSAR types [SRS_BSW_00304] do not need to support the naming convention defined in [SWS_BSW_00146].

The *BSW Module* type definitions are specified in chapter 8 of the according *BSW Module* specification. Type definitions are defined using the following template:

[SWS_BSW_00209]

Upstream requirements: SRS_BSW_00304, SRS_BSW_00009

⌈

Name:	Name of type	
Type:	Allowed entries: 'enumeration', 'structure', 'reference to' (pointer) a type, allowed AUTOSAR integer data types (SRS_BSW_00304)	
Range:	Range of legal values	Meanings, units, etc..
Description:	Informal description of the use of this type.	
Constants of this type: (optional)	Predefined names of this type.	
Available via:	Header which defines this type.	

⌋

To avoid double and inconsistent definition of data types in both *BSW Module* and Software Components, common data types are defined in *RTE Types* header files. See also [SWS_BSW_00023].

The types from the service interface chapter are provided by *Rte*, and other module types are provided by the module itself.

[SWS_BSW_00147] Definition of data types used in *Standardized Interfaces* and *Standardized AUTOSAR Interfaces*

Upstream requirements: SRS_BSW_00447

⌈Data types used in *Standardized Interface* and *Standardized AUTOSAR Interface* shall only be defined in *RTE Types* header file (`Rte_Type.h`).⌋

8.3 Function definitions

8.3.1 General specification on API functions

The function definitions for this module are specified in chapter 8 of the according *BSW Module* specification. These functions are defined using the following template:

Service name:	Name of API call	
Syntax:	Syntax of call including return type and parameters.	
Service ID [hex]:	This is the ID of service. Numbering starts for each BSW Module at 0x00. This ID is used as parameter for the error report API of Default Error Tracer	
Sync/Async:	Behavior of this service. One of these values: "Synchronous" or "Asynchronous" or "Depends on configuration"	
Reentrancy:	Reentrant / Non Reentrant	
Parameters (in):	Parameter 1	Description of parameter 1
	Parameter 2	Description of parameter 2
Parameters (inout):	Parameter 3	Description of parameter 3
Parameters (out):	Parameter 4	Description of parameter 4
Return value:	Range of legal values	Description and the circumstances under which that value is returned, and the values of configuration attributes in which the value can be returned
Description:	Short description of the API call	
Available via:	Header which makes the prototype of the API available.	

Table 8.2: General specification on API functions template

Reentrancy terms and definitions:

- **Concurrency safe:** Unlimited concurrent execution of this interface is possible, including preemption and parallel execution on multi core systems.
- **Reentrant:** Pseudo-concurrent execution (i.e. preemption) of this interface is possible on single core systems.
- **Not reentrant:** Concurrent execution of this interface is not possible.
- **Conditionally reentrant:** Concurrent execution of this interface may be possible under certain conditions. These conditions are part of API specification.

From module perspective the reentrancy attribute of APIs and callbacks means *provided reentrancy*, i.e. it is a requirement for the implementer of the module. For callouts however it is understood as *required reentrancy*, which has to be satisfied by the implementer of the callout code.

Please note that the implementation of a module entity shall be *concurrency safe* whenever its implemented entry is reentrant **and** the function is supposed to be executed on a multi-partitioned system.

The following reentrancy techniques are suggested:

- Avoid use of static and global variables
- Guard static and global variables using blocking mechanisms

- Use dynamic stack variables

The *Available via* field in the given API (or type) table shall either contain the header file which provides the prototype (or type). If the header depends on conditions or configuration parameters then this information should be given. Goal should be that the reader of the table understands how to get the required information (API prototype, type).

To avoid name clashes, all modules API functions have unique names. The *Module implementation prefix* is part of API functions name, what also eases the code reading, as every API shows to which module it belongs.

Note that the *Module implementation prefix* includes additional information from *BSW Module* provider in case of BSW Driver modules. This information is also part of the modules API names [SWS_BSW_00102].

For instance, the following API names are defined:

- `Eep_21_LDExt_Init() /* BSW Driver API */`
- `Can_21_Ext_TransmitFrame()`
- `Com_DeInit()`

[SWS_BSW_00186] Input Pointer Parameters

Upstream requirements: SRS_BSW_00486, SRS_BSW_00485

⌈All input parameters which are passed as pointers shall use the type qualifier "`const`".⌋

For example:

```
Std_ReturnType <Mip>_DoWithInputBuffer (void* Buffer)
```

Shall be changed to

```
Std_ReturnType <Mip>_DoWithInputBuffer (
    const void* Buffer)
```

[SWS_BSW_00049] Implement API parameter checking

Upstream requirements: SRS_BSW_00323, SRS_BSW_00414

⌈If the detection of *Development errors* is active for this *BSW Module* (see [SWS_BSW_00042]), then parameter checking for all API services shall be enabled.⌋

Details about API parameter checking and which results to a development error (e.g. `NULL_PTR`) and which to a runtime error (e.g. PduId range) are available in the according *BSW Module* specifications.

[SWS_BSW_00212] NULL pointer checking

Upstream requirements: SRS_BSW_00480, SRS_BSW_00337

⌈If the detection of development errors is active for this *BSW Module* (see [SWS_BSW_00042]), then pointer parameters shall be checked against `NULL_PTR` unless `NULL_PTR` is explicitly allowed as a valid pointer address value in the API parameter specification. The same also applies in case a structure address is passed for the structure's field(s).If such a violation is detected a development error shall be raised.⌋

Examples for legal `NULL_PTR` parameters are the configuration pointers for pre-compile variants in the `<Mip>_Init` functions, `PduInfoPtr->SduDataPtr` in Copy RxData and CopyTxData with `SduLength` set to zero, or the `RetryInfoPtr` in Copy TxData if retry is not supported.

The *BSW Module* API is further specified in chapter 8 of the according *BSW Module* specification.

8.3.2 Initialization function

When the *BSW Module* needs to initialize variables and hardware resources, this is done in a separate *Initialization function*. This section contains general requirements valid for all module specific implementations of an *Initialization function* service.

The *Initialization function* API name follows [1, SRS_BSW_00310] and has `Init` as *Service name*.

Examples:

- `CanIf_Init()`
- `Eep_21_LDExt_Init()`

Not all *BSW Module* have an Initialization function. Refer to chapter 7 and 8 of the according *BSW Module* specification for further details.

To protect the system against faulty initialization of the ECU or parts of the BSW, the usage of the *Initialization function* of a *BSW Module* is restricted.

[SWS_BSW_00150] Call to *Initialization functions* is restricted

Upstream requirements: SRS_BSW_00101, SRS_BSW_00467

⌈Only the *ECU State Manager* and *Basic Software Mode Manager* are allowed to call Initialization functions.⌋

The Initialization function is responsible to set the selection of configuration parameters for the module. This selection is passed as argument to the function by *ECU State Manager (EcuM)* or by the *Basic Software Mode Manager (BswM)*. See also [SWS_BSW_00050].

[SWS_BSW_00050] Check parameters passed to *Initialization functions*

Upstream requirements: SRS_BSW_00414, SRS_BSW_00400, SRS_BSW_00438

⌈If the parameter checking for the *Initialization function* is enabled [SWS_BSW_00049], the *Configuration pointer* argument shall be checked. The *Initialization function* needs a valid pointer (different from `NULL_PTR`) if either:

- The configuration can be loaded (`EcucModuleDef.postBuildLoadableSupport` is `true`)

or

- Different configuration sets can be selected (`EcucModuleDef.postBuildVariantSupport` is `true`).

If these conditions are not satisfied and *development error* checking is enabled, a *Development error* with type "Invalid configuration set selection" shall be reported to *Default Error Tracer (Det)*.⌋

See chapter 7, *Error classification*, of the according *BSW Module* specification for additional information about this error - for instance, the `Error ID`.

[SWS_BSW_00071] Module state after *Initialization function*

Upstream requirements: SRS_BSW_00450

⌈The state of a *BSW Module* shall be set accordingly at the end of *Initialization function*.⌋

Note:
This is used for *Development errors* detection

[SWS_BSW_00230] Call to *Initialization functions*

Upstream requirements: SRS_BSW_00406

⌈After a reset/reboot the module *initialization function* shall be called before any other module function. There are some module specific exceptions, e.g. pre-Init in *Dem* is always possible.⌋

[SWS_BSW_00231] Multiple calls to *Initialization functions*

Upstream requirements: SRS_BSW_00406

⌈The module *initialization function* shall not be called more than one time (see [SWS_BSW_00264]). The *initialization function* shall be called only after a reset/reboot or after a call of the modules *De-Initialization function*.⌋

8.3.3 De-Initialization function

When the *BSW Module* needs to perform functionality during ECU shutdown, change to sleep and similar phases, this is in general done in a separate *De-initialization func-*

tion. This section contains general requirements valid for all module specific implementations of a *De-initialization function* service.

The *De-initialization* function API name follows [1, SRS_BSW_00310] and has `DeInit` as *Service name*.

Example:

The AUTOSAR *COM* modules function `Com_DeInit()` stops all started *I-PDU* groups.

To protect the system against faulty de-initialization of the ECU or parts of the BSW, the usage of the *De-Initialization function* of a *BSW Module* is restricted.

[SWS_BSW_00152] Call to *De-Initialization* functions is restricted

Upstream requirements: SRS_BSW_00467

⌈Only the *ECU State Manager* and *Basic Software Mode Manager* are allowed to call *De-Initialization functions*.⌋

[SWS_BSW_00072] Module state after *De-Initialization function*

Upstream requirements: SRS_BSW_00450

⌈The state of a *BSW Module* shall be set accordingly at the beginning of the *De-Initialization function*.⌋

Note:
This is used for *Development errors* detection

[SWS_BSW_00232] Call to *De-Initialization functions*

Upstream requirements: SRS_BSW_00336

⌈The module *De-Initialization function* shall be called only if the module was initialized before (initialization function was called).⌋

[SWS_BSW_00233] Multiple calls to *De-Initialization functions*

Upstream requirements: SRS_BSW_00336

⌈The module *De-Initialization function* shall not be called more than one time after the module initialization function was called.⌋

Not all *BSW Module* have a *De-Initialization function*. Refer to chapter 7 and 8 of the according *BSW Module* specification for further details.

8.3.4 Callback notifications

[SWS_BSW_00180] Signature of *Callback functions* of *AUTOSAR Services*

Upstream requirements: SRS_BSW_00440

⌈If the *BSW Module* provides *Callback functions* which are *AUTOSAR Services*, i.e. the function invocation is routed via *RTE*, then the signature of these functions shall follow the signature provided by the *RTE* to invoke servers via `RTE_Call` *API*.⌋

If *Callback functions* do serve as simple triggers, no parameter is necessary to be passed. If additional data is to be passed to the caller within the callback scope, it must be possible to forward the content of that data using a parameter.

[SWS_BSW_00173] *Callback* parameters

Upstream requirements: SRS_BSW_00360

⌈*Callback functions* are allowed to have parameters.⌋

Some *Callback functions* are called in interrupt context. According to [SRS_BSW_00333] the *BSW Module* specification contains the information, for each *Callback function*, if it is called in interrupt context or not. The implementation of *Callback functions* called in interrupt context must be kept as short as possible, as specified in [SWS_BSW_00167].

Example: A callback from CAN Interface could be called from an ISR of the CAN driver. In this case, this information is part of the callback specification within the SWS for the CAN Interface module.

The list of callbacks is specific for every *BSW Module*. Please refer to the respective *BSW Module* specification for further details.

[SWS_BSW_00218] Usage of *Callback functions* of AUTOSAR Services

Upstream requirements: SRS_BSW_00457

⌈A *BSW Module* shall not call *RTE* interfaces (e.g. `Rte_Call`) before the first invocation of the own *MainFunction*.⌋

8.4 Scheduled functions

Many *BSW Modules* have one or more *Scheduled Functions* (also called *Main processing functions*) that have to be called cyclically or upon an event (e.g. within an OS Task) and that do the main work of the module.

Scheduled functions are directly called by *Basic Software Scheduler*. They have no return value and no parameter. Calling of *Scheduled functions* is restricted to the *BSW Scheduler*, see chapter 7.1.11.

The according *BSW Module* specification either defines one *Scheduled function* and handles all the processing internally or defines multiple *Scheduled functions* with appropriate module specific extensions. This depends on specific *BSW Module* requirements.

Scheduled functions are specified in chapter 8 of the corresponding *BSW Module* specification. These functions are defined using the following template:

Service name:	Name of API call
Syntax:	Syntax of call including return type and parameters.
Service ID[hex]:	Number of service ID. This ID is used as parameter for the error report API of *Default Error Tracer*.
Description:	Short description of the scheduled function

Table 8.3: Scheduled functions template

[SWS_BSW_00153] Naming convention for *Scheduled functions*

Upstream requirements: SRS_BSW_00373, SRS_BSW_00347

⌈*Scheduled functions* of a *BSW Module* shall be named according to the following:

`<Mip>_MainFunction[<Sd>]`

Where `<Mip>` is the *Module implementation prefix* [SWS_BSW_00102]. The content between brackets shall be used only if the module defines more than one *Scheduled function*, where `<Sd>` is a module specific name extension given to every function.⌋

Examples (for illustration only):

- Possible main processing function of *EEPROM* driver:

```
void Eep_21_LDExt_MainFunction(void)
```

- Possible main processing functions of *Communication* module:

```
void Com_MainFunctionRx(void)
void Com_MainFunctionTx(void)
void Com_MainFunctionRouteSignals(void)
```

[SWS_BSW_00154] *Scheduled functions* have no parameters

Upstream requirements: SRS_BSW_00373

⌈*Scheduled functions* shall have no parameters and no return value. Their return type is always `void`.⌋

Note:
Scheduled functions are typically not reentrant.

Scheduled functions must be able to be allocated to a basic task. Because of this, they are not allowed to enter any wait state.

[SWS_BSW_00156] *Scheduled functions* do not enter a wait state

Upstream requirements: SRS_BSW_00424

⌈*Scheduled functions* shall not enter any wait state.⌋

Typically, basic tasks are more efficient then extended tasks. Extended and basic task are classified in the *Specification of Operating System* [12].

The scheduling strategy that is built inside the *BSW Modules* must be properly documented, see also [SWS_BSW_00054].

8.5 Expected Interfaces

8.5.1 Mandatory Interfaces

The list of mandatory interfaces is specific for every *BSW Module*. Please refer to the corresponding *BSW Module* specification. These interfaces are defined using the following template:

API function	*Available via*	*Description*
Mip_APIname	Header which makes the prototype of the mandatory interface available.	Description of the API
...	...	...

Table 8.4: Mandatory interfaces template

8.5.2 Optional Interfaces

The list of optional interfaces is specific for every *BSW Module*. Please refer to the corresponding *BSW Module* specification. These interfaces are defined using the following template:

API function	*Available via*	*Description*
Mip_APIname	Header which makes the prototype of the optional interface available.	Description of the API
...	...	...

Table 8.5: Optional interfaces template

8.5.3 Callout interfaces

This chapter contains the non-configurable interfaces (*APIs*) which are expected by the module and which are provided by integration code (not by other *BSW modules*). The module will use these *APIs* in specific circumstances.

8.5.4 Configurable interfaces

Please refer to the corresponding *BSW Module* specification. In this chapter, all interfaces are listed where the target function could be configured. The target function is usually a callback function. The name of this kind of interfaces is not fixed because they are configurable. These interfaces are defined using the following template:

Service name:	Name of API call	
Syntax:	Syntax of call including return type and parameters.	
Service ID [hex]:	This is the ID of service. Numbering starts for each *BSW Module* at 0x00. This ID is used as parameter for the error report API of *Default Error Tracer*	
Sync/Async:	Behavior of this service. One of these values: "Synchronous" or "Asynchronous" or "Depends on configuration"	
Reentrancy:	Reentrant / Non Reentrant	
Parameters (in):	Parameter 1	Description of parameter 1
	Parameter 2	Description of parameter 2
Parameters (inout):	Parameter 3	Description of parameter 3
Parameters (out):	Parameter 4	Description of parameter 4
Return value:	Range of legal values	Description and the circumstances under which that value is returned, and the values of configuration attributes in which the value can be returned
Description:	Short description of the API call	
Available via:	Header which makes the prototype of the API available.	

Table 8.6: Configurable interfaces template

8.6 Service Interfaces

[SWS_BSW_00238] *ModeDeclarationGroups* definition in *BSWMD*

Upstream requirements: SRS_BSW_00334

⌈*AUTOSAR Service*, *ECU Abstraction* and *Complex Driver Components* that define a *ModeDeclarationGroupPrototype* as a *providedModeGroup* in their *BSWMD* shall define a *synchronizedModeGroup* in their *SwcBswMapping* referencing:

- The *ModeDeclarationGroupPrototype* of the *providedModeGroup*
- The corresponding *ModeDeclarationGroupPrototype* of the *ModeSwitchInferface* defined in its *SWCD*

⌋

9 Sequence diagrams

Please refer to according *BSW Module* specification.

10 Configuration specification

This chapter complements chapter 10 of according *BSW Module* specification.

10.1 Introduction to configuration specification

In addition to this section, it is highly recommended to read the documents:

- *AUTOSAR Layered Software Architecture* [4]
- *AUTOSAR ECU Configuration Specification* [6]
- This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration metamodel in detail.

The following is only a short survey of the topic and it will not replace the *ECU Configuration Specification* document.

10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic parts of an implementation of a *BSW Module*. This means that only generic or configurable module implementation can be adapted to the environment (software and hardware) in use during system and ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term "configuration class" of a parameter is used in order to refer to a specific configuration point in time.

Different configuration classes will result in different implementations and design processes, as specified in this document and in the *BSW Module* own specification.

10.1.2 Variants

Variants describe sets of configuration parameters.

In one variant, a parameter can only be of one configuration class.

[SWS_BSW_00237] *Configuration variants*

Upstream requirements: SRS_BSW_00396

⌈Different use cases require different kinds of configurability. Therefore, the following *configuration variants* are provided:

- Pre-compile: Allows individual configuration parameters to be realized at *"Pre-compile time"* only.

- Link-time: Allows individual configuration parameters to be realized at either *"Pre-compile time"* or *"Link time"*.
- Post-Build Loadable: Allows individual configuration parameters to be realized at either *"Pre-compile time"*, *"Link time"* or *"Post-build loadable time"*.
- Post-Build Selectable: Allows individual configuration parameters to be realized at either *"Pre-compile time"*, *"Link time"* or *"Post-build selectable time"*.

⌋

10.1.3 Containers

Containers hold a set of configuration parameters. This means:

- All configuration parameters are kept in containers.
- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

Configuration parameters are clustered into a container whenever:

- The configuration parameters logically belong together (e.g., general parameters which are valid for the entire module *NVRAM manager*)
- The configuration parameters need to be instantiated (e.g., parameters of the memory block specification of the *NVRAM manager* - those parameters must be instantiated for each memory block)

10.1.4 Configuration parameter tables

The tables for configuration parameters are divided in three sections:

- General section
- Configuration parameter section
- Section of included/referenced containers

10.1.4.1 General section:

SWS Item	Requirement ID	
Container Name	Identifies the container by a name, e.g., CanDriverConfiguration	
Description	Explains the intention and the content of the container .	
Post-Build Multiplicity	• **Loadable:** This container may have different number of instances in different post-build loadable configuration sets. • **-:** This container may NOT have different number of instances in different post-build loadable configuration sets.	• **Selectable:** This container may have different number of instances in different post-build selectable configuration sets. • **-:** This container may NOT have different number of instances in different post-build selectable configuration sets.
Configuration Parameters		

Table 10.1: Configuration parameters - General Section

10.1.4.2 Configuration parameter section:

Name	Identifies the parameter by name.	
Description	Explains the intention of the configuration parameter. Dependency: Describe the dependencies with respect to the scope if known.	
Type	Specifies the type of the parameter (e.g., uint8..uint32) if possible or mark it "–".	
Unit	Specifies the unit of the parameter (e.g., ms) if possible or mark it "–"	
Range	Specifies the range (or possible values) of the parameter (e.g., [1..15], ON,OFF) if possible or mark it "–".	Describes the value(s) or ranges.
Post-Build Multiplicity	• **Loadable:** This parameter may have different number of instances in different post-build loadable configuration sets. • **-:** This parameter may NOT have different number of instances in different post-build loadable configuration sets.	• **Selectable:** This parameter may have different number of instances in different post-build selectable configuration sets. • **-:** This parameter may NOT have different number of instances in different post-build selectable configuration sets.
Post-Build Value	• **Loadable:** This parameter may have different value in different post-build loadable configuration sets. • **-:** This parameter may NOT have different value in different post-build loadable configuration sets.	• **Selectable:** This parameter may have different value in different post-build selectable configuration sets. • **-:** This parameter may NOT have different value in different post-build selectable configuration sets.

Table 10.2: Configuration parameter section

10.1.4.3 Section of included/referenced containers:

Included Containers		
Container Name	***Multiplicity***	***Dependency***
Reference to a valid (sub)container by its name, e.g. CanController	Specifies the possible number of instances of the referenced container and its contained configuration parameters. Possible values: <multiplicity> <min_multiplicity..max_multiplicity>	Describes the dependencies if known or mark it as "- -".

Table 10.3: Section of included/referenced containers

10.2 General configuration specification

10.2.1 Configuration files

See chapter 5.1 for more information about the configuration file structure.

[SWS_BSW_00157] Configuration files shall be human-readable

Upstream requirements: SRS_BSW_00160

⌈Files holding configuration data for the *BSW Module* shall have a format that is readable and understandable by human beings.⌋

10.2.2 Implementation names for configuration parameters

Configuration parameters' names are specified in chapter 10 of the according *BSW Module* specification.

Example:

Name	EepNormalWriteBlockSize {EEP_NORMAL_WRITE_BLOCK_SIZE}
Description	Number of bytes written within one job processing cycle in normal mode. Implementation Type: Eep_LengthType.

Table 10.4: Configuration parameter name example

Configuration parameter name specification: It specifies the *Configuration parameter name* of this configuration parameter object in the *AUTOSAR Model*, for instance: `EepNormalWriteBlockSize`.

The same principles used for defining the names of implementation files and API functions also apply for the naming of parameters.

Note that according to [SWS_BSW_00126] all *Configuration parameter names* shall start with the *Module abbreviation* or its capitalized form.

10.2.3 Pre-compile time configuration

[SWS_BSW_00183] *Pre-Compile time configuration*

Upstream requirements: SRS_BSW_00397

⌈The *configuration parameters* in *pre-compile time* are set before compilation starts. Thus, the related configuration must be done at source code level. *Pre-compile time configuration* allows decoupling of the static configuration from implementation.⌋

Example:

```
/* File: CanTp_Cfg.h                */
/* Pre-compile time configuration */
...
#define CANTP_USE_NORMAL_ADDRESSING STD_OFF
#define CANTP_USE_NORMAL_FIXED_ADDRESSING STD_OFF
#define CANTP_USE_EXTENDED_ADDRESSING STD_ON
...
```

```
/* File: CanTp.c                    */
...
#include "CanTp_Cfg.h"
...
#if (CANTP_USE_NORMAL_ADDRESSING == STD_OFF)
...
#endif
```

The separation of configuration dependent data at compile time furthermore enhances flexibility, readability and reduces efforts for version management, as no source code is affected.

10.2.4 Link time configuration

The usage of link time parameters allows configurable functionality in *BSW Modules* that are delivered as object code. This is common, for instance, for BSW drivers.

[SWS_BSW_00184] *Link time configuration*

Upstream requirements: SRS_BSW_00398

⌈The configuration of *BSW Modules* with link time parameters is achieved on object code basis in the stage after compiling and before linking.⌋

See also [SWS_BSW_00117].

[SWS_BSW_00056] Configuration pointer to *link-time configurable* data

Upstream requirements: SRS_BSW_00344

⌈If the *BSW Module* depends on *link-time configurable* data at runtime, then it shall use a read only reference (*Configuration pointer*) to an external configuration instance.⌋

10.2.5 Post-build time configuration

Post-build time configuration mechanism allows configurable functionality of *BSW Modules* that are deployed as object code.

[SWS_BSW_00057] Implement *Post-build configuration data structure*

Upstream requirements: SRS_BSW_00438

⌈If the *BSW Module* has *Post-build time configuration* parameters, the *post-build configuration* data shall be defined in a structure: the *Post-build configuration data structure*.⌋

[SWS_BSW_00158] Use of *Configuration pointers* to *Post-build configuration data structure* is restricted

Upstream requirements: SRS_BSW_00438

⌈The *Post-build configuration data structure* of each *BSW module* shall be pointed to by *Configuration pointers*. Only *EcuM* contains *Configuration pointers* to the *Post-build configuration data structure* of post-build configurable modules which need to be initialized before the initialization of *BswM*. The rest of the *BSW modules* are initialized via configuration pointers by *BswM*.⌋

Post-build configuration data is located in a separate segment and can be loaded independently of the actual code [3]. This is the case, for instance, for loadable *CAN* configuration. To enable this independent loading of the configuration, the memory layout of these parameters must be known:

[SWS_BSW_00160] Reference pointer to *Post-build time configurable data*

Upstream requirements: SRS_BSW_00404

⌈If the *BSW Module* operates on *post-build configuration data*, then it shall use a reference (pointer) to an external configuration instance. This reference shall be provided via the *BSW module*'s initialization function (i.e., `<Mip>_Init()`) via a const-qualified function parameter.⌋

Example:

```
/* File: ComM_PBcfg.h */
...
/* Type declaration of the Configuration Type */
struct ComM_ConfigType_Tag {
...
};
...
```

```
/* File: ComM_PBcfg.c */
#include <ComM.h>
...
/* post-build time configurable data */
const ComM_ConfigType ComM_Config =
{
...
```

```
};
...
```

```
/* File: ComM.h */
#include <ComM_PBcfg.h>
...
/* Forward declaration: */
typedef struct ComM_ConfigType_Tag ComM_ConfigType;
extern void ComM_Init(const ComM_ConfigType * ComMConfigPtr);
...
```

If the *Post-build configuration* is placed at a fixed memory location and if there are no *BSW modules* with a configuration using variations points which shall be resolved at post-build time (see section 10.3) the references can be resolved as constant pointers. In that case a fixed pointer will be passed to the *BSW module*'s initialization function. Any indirections shall be kept as simple as possible.

10.2.6 Configuration variants

Independent from the configuration classes (pre-compile, link, and post-build time), configuration variants enable the reuse of ECUs in different roles within the vehicle, depending on the selected configuration variant.

[SWS_BSW_00226] Handling of different *configuration variants*

Upstream requirements: SRS_BSW_00405, SRS_BSW_00400, SRS_BSW_00345, SRS_BSW_00404, SRS_BSW_00344

⌈Regardless of the chosen *pre-compile time*, *link time* or *post-build time* configuration of a *BSW module*, multiple *configuration variants* may exist in the same configuration which is indicated by different variation points. These variation points may either be bound at *pre-compile time*, *link time* or *post-build time*.⌋

[SWS_BSW_00227] Generation of multiple *configuration variants*

Upstream requirements: SRS_BSW_00414, SRS_BSW_00399, SRS_BSW_00380

⌈In case of variation points that are bound at *post-build time* the selection of a particular variant is possible without reprogramming the ECU. To this end several *post-build time* configuration sets (i.e., one for each *configuration variant*) are generated and loaded into the ECU.⌋

[SWS_BSW_00228] Selection/binding of the *configuration variant*

Upstream requirements: SRS_BSW_00400, SRS_BSW_00405

⌈The *EcuM* will determine (via a call to `EcuM_DeterminePbConfiguration()`) which of these *post-build time* configuration variants shall be used. Based on the used *configuration variant*, the *EcuM* will then call the *BSW modules*' initialization functions ([SWS_BSW_00050], [SWS_BSW_00150]) with a pointer to the appropriate *post-build configuration variant* for the particular *BSW module*.⌋

Example:

```
/* File: ComM_PBcfg.h */
...
/* Type declaration of the Configuration Type */
typedef struct ComM_ConfigType_Tag {
...
};
...
```

```
/* File: ComM_PBcfg.c */
#include <ComM.h>
...
/* post-build time configurable data for predefined variant "VariantA"
   */
const ComM_ConfigType ComM_Config_VariantA =
{
...
};
/* post-build time configurable data for predefined variant "VariantB"
   */
const ComM_ConfigType ComM_Config_VariantB =
{
...
};
...
```

```
/* File: ComM.h */
#include <ComM_Cfg.h>
...
/* Forward declaration: */
typedef struct ComM_ConfigType_Tag ComM_ConfigType;
extern void ComM_Init(const ComM_ConfigType * ComMConfigPtr);
...
```

10.3 Published Information

Published information contains data defined by the implementer of the *BSW Module* that does not change when the module is adapted (i.e. configured) to the actual hardware and software environment. It contains version and manufacturer information.

This is necessary to provide unambiguous version identification for each *BSW Module* and enable version cross check as well as basic version retrieval facilities. Thus, the module compatibility is always visible.

[SWS_BSW_00059] Define *Published information elements*

Upstream requirements: SRS_BSW_00402, SRS_BSW_00003, SRS_BSW_00379, SRS_BSW_00374, SRS_BSW_00318

⌈The *Published information* of the *BSW Module* shall be provided within all header files by defining pre-processor directives (`#define`) and protect them against multiple definition. The preprocessor identifier is formed in the following way:

`<MIP>_<PI>`

Where <PI> is the according *Published information element* name. The module shall provide definitions for the *Published information elements* listed in the table below. These definitions shall have values with range as specified in this table:

See Table in [SWS_BSW_00256].

The *Published information* is configured in the *BSW Module* Description [5] for this module.⌋

[SWS_BSW_00256]

Upstream requirements: SRS_BSW_00402, SRS_BSW_00003, SRS_BSW_00379, SRS_BSW_00374, SRS_BSW_00318

⌈

Published information elements		
Information element	***Type / Range***	***Information element description***
<MIP>_VENDOR_ID	#define/uint16	Vendor ID (vendorId) of the dedicated implementation of this module according to the AUTOSAR vendor list.
<MIP>_MODULE_ID	#define/uint16	Module ID of this module, as defined in Appendix "List of Basic Software Modules".
<MIP>_AR_RELEASE_MAJOR_VERSION	#define/uint8	Major version number of AUTOSAR release on which the appropriate implementation is based on.
<MIP>_AR_RELEASE_MINOR_VERSION	#define/uint8	Minor version number of AUTOSAR release on which the appropriate implementation is based on.
<MIP>_AR_RELEASE_REVISION_VERSION	#define/uint8	Revision version number of AUTOSAR release on which the appropriate implementation is based on.
<MIP>_SW_MAJOR_VERSION	#define/uint8	Major version number of the vendor specific implementation of the module. The numbering is vendor specific.
<MIP>_SW_MINOR_VERSION	#define/uint8	Minor version number of the vendor specific implementation of the module. The numbering is vendor specific.
<MIP>_SW_PATCH_VERSION	#define/uint8	Patch level version number of the vendor specific implementation of the module. The numbering is vendor specific.

⌋

[SWS_BSW_00161] Restriction to declaration of vendor identification

Upstream requirements: SRS_BSW_00374

⌈The vendor identification shall be declared only in the following way, without any cast, to allow verification in a pre-processor.

```
#define <MIP>_VENDOR_ID <vi>
```

Where <vi> is the corresponding Vendor Id, as required in [SWS_BSW_00059].⌋

The following example shows the declaration of *Published information* for the CAN module implementation version 1.2.3 of vendor 43 developed according to AUTOSAR Release 4.0.3. The module ID is obtained from Chapter 11 “List of Basic Software Modules”.

Example:

```
/* File: CanIf.h */
...
/* Published information */
#define CANIF_MODULE_ID 0x003Cu
#define CANIF_VENDOR_ID 0x002Bu
#define CANIF_AR_RELEASE_MAJOR_VERSION 0x04u
#define CANIF_AR_RELEASE_MINOR_VERSION 0x00u
#define CANIF_AR_RELEASE_REVISION_VERSION 0x03u
#define CANIF_SW_MAJOR_VERSION 0x01u
#define CANIF_SW_MINOR_VERSION 0x02u
#define CANIF_SW_PATCH_VERSION 0x03u
```

Note that the *Published information elements* <MIP>_SW_MAJOR_VERSION, <MIP>_SW_MINOR_VERSION and <MIP>_SW_PATCH_VERSION are defined by software vendor.

[SWS_BSW_00162] Convention for version numbers

Upstream requirements: SRS_BSW_00321

⌈The version numbers of successive *BSW Module* implementations shall be enumerated according to the following rules:

- Increasing a more significant digit of a version number resets all less significant digits.
- The <MIP>_SW_PATCH_VERSION is incremented if the module is still upwards and downwards compatible (e.g. bug fixed)
- The <MIP>_SW_MINOR_VERSION is incremented if the module is still downwards compatible (e.g. new functionality added)
- The <MIP>_SW_MAJOR_VERSION is incremented if the module is not compatible any more (e.g. existing API changed)

The digit <MIP>_SW_MAJOR_VERSION is more significant than <MIP>_SW_MINOR_VERSION, which is more significant than <MIP>_SW_PATCH_VERSION.⌋

Example:

Take an ADC module implementation with version 1.14.2. Then:

- Versions 1.14.2 and 1.14.9 are exchangeable.
- Version 1.14.2 may contain bugs which are corrected in 1.14.9
- Version 1.14.2 can be used instead of 1.12.0, but not vice versa
- Version 1.14.2 cannot be used instead of 1.15.4 or 2.0.0

11 List of Basic Software Modules

11.1 Libraries

[SWS_BSW_91000] List of Libraries ⌈

Library name	*Library (module) abbreviation (API service prefix)*	*Library (module) ID (uint16)*
BFx Library	Bfx	205
BSW Multicore Library	Bmc	214
CRC Library	Crc	201
E2E Library	E2E	207
EFx Library	Efx	208
IFl Library	Ifl	209
IFx Library	Ifx	213
Memory Library	Msf	202
MFl Library	Mfl	210
MFx Library	Mfx	211

⌋

11.2 Modules

[SWS_BSW_91001] List of Basic Software Modules ⌈

Module name	*Module abbreviation (API service prefix)*	*Module ID (uint16)*	*AUTOSAR SW Layer*
ADC Driver	Adc	123	I/O Drivers
AUTOSAR Run-Time Interface	Arti	005	System Services
BSW Mode Manager	BswM	042	System Services
BSW Scheduler Module	SchM	130	System Services
Bulk NvData Manager	BndM	023	Memory Services
Bus Mirroring	Mirror	048	Communication Services
CAN Driver	Can	080	Communication Drivers
CAN Interface	CanIf	060	Communication Hardware Abstraction
CAN Network Management	CanNm	031	Communication Services
CAN State Management	CanSM	140	Communication Services
CAN Transceiver Driver	CanTrcv	070	Communication Hardware Abstraction
CAN Transport Layer	CanTp	035	Communication Services
CAN XL Driver	CanXL	085	Communication Drivers
CAN XL Transceiver Driver	CanXLTrcv	072	Communication Hardware Abstraction
Charging Manager	ISO15118 Chrg	215	Communication Services
COM	Com	050	Communication Services

▽

△

Module name	*Module abbreviation (API service prefix)*	*Module ID (uint16)*	*AUTOSAR SW Layer*
COM Based Transformer	ComXf	175	Communication Services
COM Manager	ComM	012	System Services
Complex Drivers	Cdd	255; 2048-4095	Complex Drivers
Core Test	CorTst	103	Microcontroller Drivers
Crypto Driver	Crypto	114	Crypto Drivers
Crypto Interface	CryIf	112	Crypto Hardware Abstraction
Crypto Service Manager	Csm	110	Crypto Services
Data Distribution Service	Dds	047	Communication Services
Data Distribution Service Transformer	DdsXf	187	Communication Services
Default Error Tracer	Det	015	System Services
Diagnostic Communication Manager	Dcm	053	Communication Services
Diagnostic Event Manager	Dem	054	System Services
Diagnostic Log and Trace	Dlt	055	Communication Services
Diagnostic over IP	DoIP	173	Communication Services
DIO Driver	Dio	120	I/O Drivers
DMA Driver	Dma	216	Microcontroller Drivers
E2E Transformer	E2EXf	176	Communication Services
ECU State Manager	EcuM	010	System Services
EEPROM Abstraction	Ea	040	Memory Hardware Abstraction
Ethernet Driver	Eth	088	Communication Drivers
Ethernet Interface	EthIf	065	Communication Hardware Abstraction
Ethernet State Manager	EthSM	143	Communication Services
Ethernet Switch Driver	EthSwt	089	Communication Hardware Abstraction
Ethernet Transceiver Driver	EthTrcv	073	Communication Hardware Abstraction
Firewall	Fw	111	Communication Services
Flash EEPROM Emulation	Fee	021	Memory Hardware Abstraction
Flash Test	FlsTst	104	Memory Drivers
FlexRay Driver	Fr	081	Communication Drivers
FlexRay Interface	FrIf	061	Communication Hardware Abstraction
FlexRay ISO Transport Layer	FrTp	036	Communication Services
FlexRay Network Management	FrNm	032	Communication Services
FlexRay State Manager	FrSM	142	Communication Services
FlexRay Transceiver Driver	FrTrcv	071	Communication Hardware Abstraction
Function Inhibition Manager	FiM	011	System Services
GPT Driver	Gpt	100	Microcontroller Drivers
I2C Driver	I2C	113	Communication Drivers
ICU Driver	Icu	122	I/O Drivers
IEEE1722 Transport Layer	IEEE1722Tp	131	Communication Services
Intrusion Detection System Manager	IdsM	108	Crypto Services
IO HW Abstraction	IoHwAb	254	I/O Hardware Abstraction
IPDU Multiplexer	IpduM	052	Communication Services
Key Manager	KeyM	109	Crypto Services
LIN Driver	Lin	082	Communication Drivers

▽

△

Module name	*Module abbreviation (API service prefix)*	*Module ID (uint16)*	*AUTOSAR SW Layer*
LIN Interface	LinIf	062	Communication Hardware Abstraction
LIN State Manager	LinSM	141	Communication Services
LIN Transceiver Driver	LinTrcv	064	Communication Hardware Abstraction
LSDU Router	LSduR	132	Communication Hardware Abstraction
MACsec Key Agreement	Mka	151	Communication Services
MCU Driver	Mcu	101	Microcontroller Drivers
Memory Abstraction Interface	MemIf	022	Memory Hardware Abstraction
Memory Access	MemAcc	041	Memory Hardware Abstraction
Memory Driver	Mem	091	Memory Drivers
Network Management Interface	Nm	029	Communication Services
NVRAM Manager	NvM	020	Memory Services
OCU Driver	Ocu	125	I/O Drivers
OS	Os	001	System Services - OS
PDU Router	PduR	051	Communication Services
Port Driver	Port	124	I/O Drivers
PWM Driver	Pwm	121	I/O Drivers
RAM Test	RamTst	093	Memory Drivers
RTE	Rte	002	RTE
SAE J1939 Diagnostic Communication Manager	J1939Dcm	058	Communication Services
SAE J1939 Functional Safety Communication Protocol	J1939Fscp	194	Communication Services
SAE J1939 Network Management	J1939Nm	034	Communication Services
SAE J1939 Request Manager	J1939Rm	059	Communication Services
SAE J1939 Transport Layer	J1939Tp	037	Communication Services
Secure Onboard Communication	SecOC	150	Communication Services
Service Discovery	Sd	171	Communication Services
Socket Adaptor	SoAd	056	Communication Services
Software Cluster Connection	SwCluC	016	System Services
SOME/IP Transformer	SomeIpXf	174	Communication Services
SOME/IP Transport Protocol	SomeIpTp	177	Communication Services
SPI Handler Driver	Spi	083	Communication Drivers
Synchronized Time-Base Manager	StbM	160	System Services
TCP/IP Stack	TcpIp	170	Communication Services
Time Service	Tm	014	System Services
Time Sync over CAN	CanTSyn	161	Communication Services
Time Sync over Ethernet	EthTSyn	164	Communication Services
Time Sync over FlexRay	FrTSyn	163	Communication Services
UDP Network Management	UdpNm	033	Communication Services
VDP Communication Module Remote	VdpCmR	188	Communication Services
Watchdog Driver	Wdg	102	Microcontroller Drivers
Watchdog Interface	WdgIf	043	Onboard Device Abstraction
Watchdog Manager	WdgM	013	System Services

▽

△

Module name	*Module abbreviation (API service prefix)*	*Module ID (uint16)*	*AUTOSAR SW Layer*
XCP	Xcp	212	Communication Services

⌋

11.3 Special Files

[SWS_BSW_91002] List of special files ⌈

AUTOSAR name	*Short name (API service prefix)*	*File name*	*Module ID*
Communication Stack Types	Comtype	ComStack_Types.h	196
Memory Mapping	MemMap	(Prefix_)MemMap.h	195
Platform Types	Platform	Platform_Types.h	199
Standard Types	Std	Std_Types.h	197

⌋

A Change History

Please note that the lists in this chapter also include constraints and specification items that have been removed from the specification in a later version. These constraints and specification items do not appear as hyperlinks in the document.

A.1 Change History of this document according to AUTOSAR Release R22-11

A.1.1 Added Specification Items in R22-11

[SWS_BSW_00255]

A.1.2 Changed Specification Items in R22-11

[SWS_BSW_00010] [SWS_BSW_00042] [SWS_BSW_00136] [SWS_BSW_00144] [SWS_BSW_00186]

A.1.3 Deleted Specification Items in R22-11

[SWS_BSW_00009] [SWS_BSW_00132] [SWS_BSW_00178] [SWS_BSW_00187] [SWS_BSW_00202] [SWS_BSW_00235]

A.2 Change History of this document according to AUTOSAR Release R23-11

A.2.1 Added Specification Items in R23-11

[SWS_BSW_00256] [SWS_BSW_00257]

A.2.2 Changed Specification Items in R23-11

none

A.2.3 Deleted Specification Items in R23-11

[SWS_BSW_00039]

A.3 Change History of this document according to AUTOSAR Release R24-11

A.3.1 Added Specification Items in R24-11

[SWS_BSW_00258]

A.3.2 Changed Specification Items in R24-11

[SWS_BSW_00045] [SWS_BSW_00101] [SWS_BSW_00103] [SWS_BSW_00144] [SWS_BSW_00153] [SWS_BSW_00252] [SWS_BSW_00253] [SWS_BSW_00256]

A.3.3 Deleted Specification Items in R24-11

[SWS_BSW_00223] [SWS_BSW_00224] [SWS_BSW_00225]

A.4 Change History of this document according to AUTOSAR Release R25-11

A.4.1 Added Specification Items in R25-11

[SWS_BSW_00245] [SWS_BSW_00259] [SWS_BSW_00260] [SWS_BSW_00261] [SWS_BSW_00262] [SWS_BSW_91000] [SWS_BSW_91001] [SWS_BSW_91002]

A.4.2 Changed Specification Items in R25-11

[SWS_BSW_00257]

A.4.3 Deleted Specification Items in R25-11

[SWS_BSW_00135] [SWS_BSW_00252] [SWS_BSW_00253]

A.5 Change History of this document according to AUTOSAR Release R26-11

A.5.1 Added Specification Items in R26-11

[SWS_BSW_00263] [SWS_BSW_00264] [SWS_BSW_00265] [SWS_BSW_00266] [SWS_BSW_00267]

A.5.2 Changed Specification Items in R26-11

[SWS_BSW_00046] [SWS_BSW_00050] [SWS_BSW_00066] [SWS_BSW_00115] [SWS_BSW_00143] [SWS_BSW_00144] [SWS_BSW_00204] [SWS_BSW_00205] [SWS_BSW_00230] [SWS_BSW_00237] [SWS_BSW_00243] [SWS_BSW_91001]

A.5.3 Deleted Specification Items in R26-11

[SWS_BSW_00036] [SWS_BSW_00051] [SWS_BSW_00052] [SWS_BSW_00064] [SWS_BSW_00164] [SWS_BSW_00172] [SWS_BSW_00236]