Document Title	General Specification of Basic Software Modules
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	578

Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.4.0

Document Change History

Date	Release	Changed by	Change Description
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 SupportForPBLAndPBSECUConfiguration concept • minor corrections / clarifications / editorial changes; For details please refer to the BWCStatement

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
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 Administration	• Initial release

Disclaimer

Table of Contents

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 [3].

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 [13], chapter “Support for traceability” 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 [7], 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 [3]. 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

Abbreviation / Acronym:	***Description:***
BSW driver	For a list of BSW drivers see the List of Basic Software Modules [1], 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 (see chapter 3.1): “camelCase (…) *is the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound and the first letter either upper or lower case (…).*” Example: GetVersionInfo
<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> (see bsw_constr_001) completely written in upper case.
MCAL	The MCAL, Microcontroller Abstraction Layer, is defined in AUTOSAR Layered Software Architecture [2]
<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.
WCET	Worst case execution time.
BSWMD	Basic Software Module Description
SWCD	Software Component Description

3 Related documentation

3.1 Input documents

[1] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList.pdf

[2] AUTOSAR Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[3] AUTOSAR General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral.pdf

[4] AUTOSAR Specification of BSW Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

[5] AUTOSAR Specification of RTE
AUTOSAR_SWS_RTE.pdf

[6] AUTOSAR Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

[7] AUTOSAR Glossary
AUTOSAR_TR_Glossary.pdf

[8] AUTOSAR Specification of Operating System
AUTOSAR_SWS_OS.pdf

[9] AUTOSAR Specification of Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[10] AUTOSAR Specification of Diagnostic Event Manager
AUTOSAR_SWS_DiagnosticEventManager.pdf

[11] AUTOSAR Methodology
AUTOSAR_TR_Methodology.pdf

[12] AUTOSAR Specification of Standard Types
AUTOSAR_SWS_PlatformTypes.pdf

[13] AUTOSAR Standardization Template
AUTOSAR_TPS_StandardizationTemplate.pdf

[14] AUTOSAR Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[15] AUTOSAR Specification of Default Error Tracer
AUTOSAR_SWS_ DefaultErrorTracer.pdf

[16] CamelCase – Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/CamelCase

3.2 Related standards and norms

[17] MISRA C 2012 Standard
Homepage: http://www.misra.org.uk/

[18] IEC 7498-1 The Basic Model, IEC Norm, 1994

4 Constraints and assumptions

4.1 Limitations

This specification is common to all AUTOSAR *BSW Modules* [1] 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* [1] 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 the *List of Basic Software Modules* [1] within the column “Module Abbreviation”.

[SWS_BSW_00101] *Module abbreviation*
[The *Module abbreviation* `<Ma>` of a *BSW Module* shall be the same as defined in the *List of Basic Software Modules* [1].](*SRS_BSW_00300*)

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*
[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`.](*SRS_BSW_00300*, *SRS_BSW_00347*)

The elements `vendorId` and `vendorApiInfix` are defined in *BSW Module Description Template* [4]. 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 *BSW Module Description Template* [4].

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 [4].
Implementation source	mandatory	`Com.c`
Implementation header	mandatory	`Com.h`
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 1: Module Implementation Files

Note that according to AUTOSAR Methodology [11] 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

[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 wildcards * are replaced according to the different types of files specified for the module.]*(SRS_BSW_00300)*

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
[File names shall be considered case sensitive regardless of the file system in which they are used.]*(SRS_BSW_00464)*

[SWS_BSW_00171] File names are non-ambiguous
[It shall not be allowed to name any two files so that they only differ by the case of their letters.]*(SRS_BSW_00465)*

5.1.3 Imported and exported information

[SWS_BSW_00104] Restrict imported information
[The *BSW Module* shall import only the necessary information (i.e. header files) that is required to fulfill its functional requirements.]*(SRS_BSW_00301)*

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
[The *BSW Module* shall export only that kind of information in their corresponding header files that is explicitly needed by other modules.]*(SRS_BSW_00302)*

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*
[The *BSW Module description* (`.arxml`) shall be provided for the module according to the AUTOSAR Specification of BSW Module Description Template [4].](*SRS_BSW_00423, SRS_BSW_00426, SRS_BSW_00427, SRS_BSW_00334*)

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 [4].

5.1.5 Module documentation

[SWS_BSW_00002] Provide *BSW Module documentation*
[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.](*SRS_BSW_00009, SRS_BSW_00010*)

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
[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

]*(SRS_BSW_00341)*

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*
[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).]*(SRS_BSW_00428)*

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
[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.]*(SRS_BSW_00167)*

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
[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
```

]*(SRS_BSW_00346)*

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

[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*
[The module *Implementation source* files of the *BSW Module* shall include its own *Implementation header* .]*(SRS_BSW_00346)*

The *Memory mapping header* is necessary to enable the *BSW Module* to access the module specific functionality provided by the *BSW Memory Mapping* [6].

[SWS_BSW_00006] Include *Memory mapping header*
[The *Implementation source* files of the *BSW Module* shall include the *BSW Memory mapping header* (`<Mip>_MemMap.h`).]*(SRS_BSW_00437)*

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*
[If the *BSW Module* uses BSW Scheduler API or if it implements `BswSchedulableEntity`s, 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.]*(SRS_BSW_00415)*

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 [5]. `BswSchedulableEntity`s are defined in *BSW Module Description Template* [4].

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*
[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`).]*(SRS_BSW_00409)*

For further information, see chapter 7.2 -Error Handling.

[SWS_BSW_00009] Include own *Callback header*
[If the *BSW Module* implementation contains *Callback functions*, then its *Implementation source* files shall include the *BSW Modules'* own *Callback header* .]*(SRS_BSW_00447)*

To access callbacks from other modules, the according *Callback 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 *Callback headers*
[If the *BSW Module* implementation calls *Callback functions* from other modules, then the *Implementation source* files of the *BSW Module* shall include the *Callback 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.*] *(SRS_BSW_00447)*

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
[If the *BSW Module* implements *Interrupt Service Routines*, then these routines shall be implemented in one or more of its *Implementation source* files.]*(SRS_BSW_00314)*

[SWS_BSW_00181] Implement ISRs in a separate file
[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.]*(SRS_BSW_00314)*

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
[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`

]*(SRS_BSW_00346)*

[SWS_BSW_00014] Define all *Link time configuration parameters*

[The *Link time configuration source* shall contain definitions for all link time configuration parameters specified for this module.](*SRS_BSW_00158*, SRS_BSW_00380)

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
[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
```

](*SRS_BSW_00346*)

[SWS_BSW_00063] Define all *Post-build time configuration parameters*
[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.](*SRS_BSW_00158*, *SRS_BSW_00380*)

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
[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
```

]*(SRS_BSW_00314)*

[SWS_BSW_00021] Implement *Interrupt frame routines*
[The *Interrupt frame implementation source* shall contain implementation of all *Interrupt frame* routines specified for this *BSW Module*.]*(SRS_BSW_00314)*

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*
[The *Interrupt frame implementation source* files of a *BSW Module* shall include the *Implementation Header* of this *BSW Module.*]*(SRS_BSW_00314)*

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
[The header file structure shall contain one or more files that provide the declaration of functions from the *BSW Module* 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.]*(SRS_BSW_00346)*

[SWS_BSW_00110] Content of *Implementation header*

[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*
[If the *BSW Module* implements AUTOSAR Services, then it shall include its *Application Types Header File* in its *Implementation header* file or files.]*(SRS_BSW_00447)*

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 [5] – 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 `GetErrorStatus`.

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*
[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`).]*(SRS_BSW_00348)*

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

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

[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_00018] Declare *Interrupt frame* routines
[If the *BSW Module* implements *Interrupt frame* routines (SWS_BSW_00021), then their declaration shall be done in its *Implementation header* file or files.
]*(SRS_BSW_00314)*

[SWS_BSW_00043] Declare *Interrupt Service Routines*
[If the *BSW Module* implements *Interrupt Service Routines* (*ISR*), then their declaration shall be done in its *Implementation header* file or files.]*(SRS_BSW_00439)*

[SWS_BSW_00068]
Support *Interrupt Service Routines categories 1 and 2*
[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.]*(SRS_BSW_00439)*

See also chapter 7.1.15 - Interrupt service routines.

[SWS_BSW_00210] *Exclusion of MainFunction and BswModuleClientServerEntrys from the Implementation header*
[The module header files shall not include the prototype declarations of MainFunctions and BswModuleClientServerEntrys that are expected to be invoked by the RTE/BswScheduler.]*()*

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*
[If the *BSW Module* implements *AUTOSAR Services*, then it shall include its *Application Header File* in module files using RTE interfaces.]*(SRS_BSW_00447)*

[SWS_BSW_00069] Restrict inclusion for *Application Header File*
[The *Application Header File* shall not be included in *BSW Module* files that are included directly or indirectly by other modules.]*(SRS_BSW_00447)*

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* [5] – section “File Contents”, requirement [SWS_Rte_1006].

More information about the *Application Header File* can be found in the *Specification of RTE* [5] – section “Application Header File”.

Note that the application header file includes by its own the *Application Types Header File*. See *Specification of RTE* [5], [SWS_Rte_7131], and SWS_BSW_00023.

5.1.8 Version check

The integration of AUTOSAR *BSW Modules* is supported by the execution of *Inter Module Checks*: Each *BSW Module* performs a pre-processor check of the versions of all imported include files. During configuration, a methodology supporting tool checks whether the version numbers of all integrated modules belong to the same AUTOSAR major and minor release, i.e. if all modules are from the same AUTOSAR baseline. If not, an error is reported.

The execution of *Inter Module Checks* is necessary to avoid integration of incompatible modules. Version conflicts are then detected in early integration phase.

[SWS_BSW_00036] Perform *Inter Module Checks*
⌈The *BSW Module* shall perform *Inter Module Checks* to avoid integration of incompatible files: For every included header file that does not belong to this module, the following *Published information elements* (SWS_BSW_00059) shall be verified through pre-processor checks:

- Major AUTOSAR Release Number (`<MIP>_AR_RELEASE_MAJOR_VERSION`)
- Minor AUTOSAR Release Number (`<MIP>_AR_RELEASE_MINOR_VERSION`)

If the values are not identical to the values expected by the implementation of this module, an error shall be reported.⌋*(SRS_BSW_00004)*

Note: The intention of the AUTOSAR standard is to keep revisions of the same AUTOSAR Major and Minor release compatible.

6 Requirements traceability

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 [3]. 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.

Requirements traceability to document:
General Requirements on Basic Software Modules [3]

Requirement	Description	Satisfied by
SRS_BSW_00003	All software modules shall provide version and identification information	SWS_BSW_00059
SRS_BSW_00004	All Basic SW Modules shall perform a pre-processor check of the versions of all imported include files	SWS_BSW_00036
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 2012 Standard.	SWS_BSW_00115
SRS_BSW_00009	All Basic SW Modules shall be documented according to a common standard.	SWS_BSW_00002
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
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_00158	-	SWS_BSW_00014, SWS_BSW_00063
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	All AUTOSAR Basic Software Modules shall be identified by an unambiguous name	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_00105
SRS_BSW_00304	All AUTOSAR Basic Software Modules shall use the following data types instead of native C data types	SWS_BSW_00120
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	SWS_BSW_00131

	shall indicate all global data with read-only purposes by explicitly assigning the const keyword	
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
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
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_00330	It shall be allowed to use macros instead of functions where source code is used and runtime is critical	SWS_BSW_00132
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	All Basic Software Modules shall provide an XML file that contains the meta data	SWS_BSW_00001, SWS_BSW_00238
SRS_BSW_00335	Status values naming convention	SWS_BSW_00124
SRS_BSW_00337	Classification of development errors	SWS_BSW_00042, SWS_BSW_00045, SWS_BSW_00073
SRS_BSW_00339	Reporting of production relevant error status	SWS_BSW_00046, SWS_BSW_00066, SWS_BSW_00247, SWS_BSW_00248
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
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
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 integer type definitions of target and compiler specific scope shall be placed and organized in a single type header	SWS_BSW_00120
SRS_BSW_00359	All AUTOSAR Basic Software Modules callback functions shall avoid return types other than void if possible	SWS_BSW_00172
SRS_BSW_00360	AUTOSAR Basic Software Modules callback functions are allowed to have parameters	SWS_BSW_00173
SRS_BSW_00361	All mappings of not standardized keywords of compiler specific scope shall be placed and organized in a compiler specific type and keyword header	SWS_BSW_00178
SRS_BSW_00371	The passing of function pointers as API parameter is forbidden for all AUTOSAR Basic Software Modules	SWS_BSW_00149
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
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
SRS_BSW_00380	Configuration parameters being stored in memory shall be placed into separate c-files	SWS_BSW_00014, SWS_BSW_00063
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_00400	Parameter shall be selected from multiple sets of parameters after code has been loaded and started	SWS_BSW_00050, SWS_BSW_00228
SRS_BSW_00402	Each module shall provide version information	SWS_BSW_00059
SRS_BSW_00404	BSW Modules shall support post-build configuration	SWS_BSW_00160

SRS_BSW_00405	BSW Modules shall support multiple configuration sets	SWS_BSW_00228
SRS_BSW_00407	Each BSW module shall provide a function to read out the version information of a dedicated module implementation	SWS_BSW_00052, SWS_BSW_00059, SWS_BSW_00064, SWS_BSW_00164
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, SWS_BSW_00246
SRS_BSW_00410	Compiler switches shall have defined values	SWS_BSW_00123
SRS_BSW_00411	All AUTOSAR Basic Software Modules shall apply a naming rule for enabling/disabling the existence of the API	SWS_BSW_00051
SRS_BSW_00413	An index-based accessing of the instances of BSW modules shall be done	SWS_BSW_00047
SRS_BSW_00414	Init functions shall have a pointer to a configuration structure as single parameter	SWS_BSW_00049, SWS_BSW_00050
SRS_BSW_00415	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	SWS_BSW_00007
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
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
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_00009, 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
SRS_BSW_00460	Reentrancy Levels	SWS_BSW_00039
SRS_BSW_00463	Naming convention of callout prototypes	SWS_BSW_00135, SWS_BSW_00136
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_00477	The functional interfaces of AUTOSAR BSW modules shall be specified in C90	SWS_BSW_00234

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
[If the *BSW Module* implementation is written in C language, then it shall conform to the MISRA C 2012 Standard [17].]*(SRS_BSW_00007)*

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
[The external interface binding of the *BSW Modules* shall be conform to ISO/IEC 9899:1990 (C90) standard, with extensions:

- to allow use of 64bit types (i.e. long long)
- to increase the number of significant characters allowed for external identifier

] (*SRS_BSW_00477*)

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 [3].

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
[The *BSW Module* implementation shall support a tool based configuration, as described in AUTOSAR Methodology [11].]*(SRS_BSW_00159)*

For more information about ECU configuration, see also AUTOSAR Specification of ECU Configuration [14].

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
[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.]*(SRS_BSW_00342)*

7.1.4 Platform independency and compiler abstraction

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* [2]. The list of *BSW Modules* from *MCAL* is available in the *List of BSW Modules* [1]: Microcontroller Drivers, I/O Drivers, Communication Drivers and Memory Drivers.

[SWS_BSW_00119] *Platform independent BSW Modules*
[If the *BSW Module* is classified as *Platform independent BSW Module*, then its source code shall not be processor dependent.]*(SRS_BSW_00006)*

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
[The *BSW Module* implementation shall not use compiler and platform specific keywords directly.]*(SRS_BSW_00306)*

[SWS_BSW_00178] Mapping of compile specific keywords
[If the *BSW Module* implementation needs compiler specific keywords, then these keywords shall be redefined (mapped) in a separate file, the *Compiler Specific Language Extension Header* (`Compiler.h`).]*(SRS_BSW_00361)*

Example: Compiler specific keywords can be mapped to compiler independent keywords by defining macros in `Compiler.h`:

```
/* Compiler.h      */
#define FAR(X)    __far__ X
```

This enables the usage of this macro within source code in the following way:

```
FAR(void) function();
```

In this example, the compiler dependency is encapsulated in a separate file (`Compiler.h`) which can be exchanged if a new compiler is used. This enables the provision of a compiler specific header containing proprietary pre-processor directives as well as wrapper macros for all specialized language extensions.

Note that different compilers can require extended keywords to be placed in different places. Example:

Compiler 1 requires:

```
void __far__ function();
```

Compiler 2 requires:

```
__far__ void function();
```

In this case it is not possible to accommodate the different implementations with inline macros, so a function-like macro style is adopted instead. This macro wraps

the return type of the function and therefore permits additions to be made, such as `__far__`, either before or after the return type.

Example:

Compiler 1:

```
/* Compiler.h      */
#define FAR(x) x __far__
```

Compiler 2:

```
/* Compiler.h      */
#define FAR(x) __far__ x
```

The following usage can expand to the examples given above:

```
FAR(void) function();
```

Although this last example conflicts with the MISRA Rule 20.4, see chapter 3.1, it is a reasonable solution and this exception is acceptable when necessary.

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 BSW_SWS_061

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
[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*.]*(SRS_BSW_00171)*

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, see chapter 3.1, see SWS_BSW_118. 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 [Compiler switches shall be compared with defined values. Simply checking if a compiler switch is defined shall not be used in implementation.]*(SRS_BSW_00410)*

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
[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.]*(SRS_BSW_00441, SRS_BSW_00335)*

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_DRA_CONFIG,
   EEP_21_LDEXT_ARE,
   EEP_21_LDEXT_EV
} Eep_21_LDExt_NotificationType;
```

[SWS_BSW_00125] Naming convention for *Error values*
[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.]*(SRS_BSW_00327)*

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- Implementation names.

[SWS_BSW_00126] Naming conventions for *Configuration parameters names*
[*Configuration parameter names* for configuration parameters which are not published shall be named in one of the following ways:

Camel case: `<Ma><Pn>`

If the configuration parameter is published, then one of the following conventions shall be used:

Camel case: `<Mip><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.

The usage of the *camel case* or upper case notation shall be chosen according to the original *Configuration parameter name specification.*]*(SRS_BSW_00408, SRS_BSW_00347)*

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
[The *BSW Module* implementation shall avoid duplication of code.]*(SRS_BSW_00328)*

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
[If the *BSW Module* defines global variables, then their definition shall take place in the *Implementation source* file.]*(SRS_BSW_00308)*

[SWS_BSW_00130] Naming convention for global variables
[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*.]*(SRS_BSW_00307)*

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
[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.]*(SRS_BSW_00309)*

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.

[SWS_BSW_00132] Usage of macros and inline functions
[The usage of macros and inline functions is allowed, for instance, to improve runtime behavior. It is advised to consider the MISRA-C 2012 rules with respect to INLINE functions and MACRO.]*(SRS_BSW_00330)*

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 8.5.

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* [5].

[SWS_BSW_00133] Calling *Scheduled functions* is restricted
[The *BSW Module* implementation shall not contain calls to *Scheduled functions* (*Main processing functions*).]*(SRS_BSW_00433)*

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*
[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.]*(SRS_BSW_00450)*

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*
[The *Exclusive areas* of the *BSW Module* shall be defined and documented as described in the specification of *BSW Module Description Template* [4] within the *BSW Module Description*.]*(SRS_BSW_00426)*

[SWS_BSW_00134] Restriction to usage of *Exclusive areas*
[The *Exclusive areas* of the *BSW Module* shall only protect module internal data.]*(SRS_BSW_00426)*

7.1.13 Callouts

[SWS_BSW_00039] Define prototypes of *Callout functions*
[If the *BSW Module* uses *Callout functions*, then it shall define the prototype of the callouts in its own *Implementation header*.]*(SRS_BSW_00460)*

The file containing the implementation of the *Callout function* can include this header to check if declaration and definition of callout match.

Example: Operating System

```
/* File: Os.h    */
…
/* Callout declaration */
void ErrorHook ( StatusType );
…
```

[SWS_BSW_00135] Conventions for *Callout functions* prototype declaration
[The following convention shall be used for declaration of *Callout functions* prototypes:

```
/* --- Start section definition: --- */

#define <MIP>_START_SEC_<CN>_CODE

/* --- Function prototype definition: --- */

FUNC(void, <MIP>_<CN>_CODE) <Cn> (void);

/* --- Stop section definition: --- */

#define <MIP>_STOP_SEC_<CN>_CODE
```

Where MIP is the Module implementation prefix of the calling module, `<CN>` is the *Callout name*, which shall have the same spelling of the *Callout name*, including module reference, but written in upper case and `<Cn>` is the *Callout name,* using the conventional *camel case* notation for API names.]*(SRS_BSW_00463)*

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.

[SWS_BSW_00136] Memory section and memory class of *Callout functions*
[Each *Callout function* shall be mapped to its own memory section and memory class. These memory classes will then be mapped to the actually implemented memory classes at integration time.]*(SRS_BSW_00463)*

For example:

```
#define COM_START_SEC_SOMEMODULE_SOMECALLOUT_CODE
#include "Com_MemMap.h"
FUNC(void, COM_SOMEMODULE_SOMECALLOUT_CODE)
Somemodule_SomeCallout (void);
#define COM_STOP_SEC_SOMEMODULE_SOMECALLOUT_CODE
#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
[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* [9] within the *BSW Module Description*.]*(SRS_BSW_00423)*

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 - Platform independency and compiler abstraction.

[SWS_BSW_00137] ISR implementation is platform dependent
[If the *BSW Module* is classified as *Platform independent BSW Module*, it shall not implement interrupt service routines.]*(SRS_BSW_00164)*

For more explanation on *Platform independent BSW Modules,* see the section 7.1.4 - Platform independency and compiler abstraction.

[SWS_BSW_00167] Keep runtime of ISR as short as possible

[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.]*(SRS_BSW_00325, SRS_BSW_00333)*

ISR functions are defined with a name and the category according to the AUTOSAR OS, see chapter 3.1.

[SWS_BSW_00041] Define and document ISR routines
[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* [4] within the *BSW Module Description*.]*(SRS_BSW_00427)*

[SWS_BSW_00065] Support for interrupt category CAT2
[If the *BSW Module* implements *Interrupt service routines (*ISR), then the implementation shall at least support interrupt category CAT2.]*(SRS_BSW_00427)*

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
[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*.

]*(SRS_BSW_00164)*

The definition of ISR categories CAT1 and CAT2 is available in AUTOSAR General Requirements on Basic Software Modules [3]. For more information see also the Specification of RTE [5], chapter “Interrupt decoupling and notification”.

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

OS Services	**RTE , BSW Scheduler, 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		✓			
GetApllicationID	✓				
StartScheduleTable	✓	✓			
StopScheduleTable	✓	✓			
NextScheduleTable	✓	✓			
SyncScheduleTable	✓	✓		✓	
GetScheduleTableStatus	✓	✓		✓	
SetScheduleTableAsync	✓	✓			
IncrementCounter	✓				
GetCounterValue	✓	✓	✓	✓	✓
GetElapsedCounterValue	✓	✓	✓	✓	✓
TerminateApplication	✓				

OS Services	RTE , BSW Scheduler, BswM, CDD	EcuM	MCAL	StbM	Other *BSW Modules*
AllowAccess	✓				
GetApplicationState	✓				
ControlIdle	✓	✓			
GetNumberOfActivatedCores	✓				
GetCoreID	✓	✓	✓	✓	✓
StartCore		✓			
StartNonAutosarCore		✓			
GetSpinlock	✓	✓	✓		
ReleaseSpinlock	✓	✓	✓		
TryToGetSpinlock	✓	✓	✓		
ShutdownAllCores		✓			
ReadPeripheral8	✓		✓		
ReadPeripheral16	✓		✓		
ReadPeripheral32	✓		✓		
WritePeripheral8	✓		✓		
WritePeripheral16	✓		✓		
WritePeripheral32	✓		✓		
ModifyPeripheral8	✓		✓		
ModifyPeripheral16	✓		✓		
ModifyPeripheral32	✓		✓		
DisableInterruptSource	✓		✓		
EnableInterruptSource	✓		✓		
ClearPendingInterrupt	✓		✓		
ActivateTaskAsyn	✓				
SetEventAsyn	✓				

Table 2: OS Services and associated permissions

.]*(SRS_BSW_00429)*

The according services are described in AUTOSAR OS.

7.1.17 Access to hardware registers

[SWS_BSW_00179] Concurrent access to registers
[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](*SRS_BSW_00451*)

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
[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`)
- includes the *Compiler Specific Language Extension Header* (`Compiler.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* [12].

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
[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`)](*SRS_BSW_00304*, *SRS_BSW_00353*)

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:

1. Fixed size guaranteed:

Data type	Representation
`uint8`	8 bit
`uint16`	16 bit
`uint32`	32 bit
`sint8`	7 bit + 1 bit sign
`sint16`	15 bit + 1 bit sign
`sint32`	31 bit + 1 bit sign

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

Data type	Representation
`uint8_least`	At least 8 bit
`uint16_least`	At least 16 bit
`uint32_least`	At least 32 bit
`sint8_least`	At least 7 bit + 1 bit sign
`sint16_least`	At least 15 bit + 1 bit sign
`sint32_least`	At least 31 bit + 1 bit sign

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:

```
FALSE = 0
TRUE  = 1
```

[SWS_BSW_00142] Allowed operations with `boolean` variables
[The only allowed operations with variables from type `boolean` are: assignment, return and test for equality, inequality and logical not with `TRUE` or `FALSE`.]*(SRS_BSW_00378)*

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

1. the RTE transports the service call to the partition where the BSW module entity that shall execute the call is located, or
2. 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
⌈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
⌈ 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
⌈ 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.

7.2 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
⌈All errors, which may be detected and/or reported by a *BSW Module*, are classified in six different types:

- development errors [SRS_BSW_00337]
- runtime errors [SRS_BSW_00452]
- transient faults [SRS_BSW_00473]
- production errors [SRS_BSW_00458]
- extended production errors [SRS_BSW_00466]
- security event [SRS_CryptoStack_00122]

⌋

7.2.2 Development errors

7.2.2.1 Synopsis

Development errors 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.

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
[*Development error* values are of type *uint8.*]*()*

7.2.2.3 Configuration

[SWS_BSW_00202] Activation of *Development Errors*
[The activation of *development errors* is done via an C pre-processor switch. The switch `<Ma>DevErrorDetect` shall activate or deactivate the detection of all *development errors* of a module.]*()*

[SWS_BSW_00203] API parameter checking
[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*
[The detection and reporting of *Development errors* shall be performed only if the configuration parameter for detection of *Development errors* is set.]
(SRS_BSW_00337).

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”).

[SWS_BSW_00235] Default configuration value of *Development errors*
[The detection and reporting of *Development errors* shall be configurable and the default value of the configuration shall be that those error type is disabled.] ()

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*

[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 *List of BSW Modules* [1]]) to identify itself.]*(SRS_BSW_00337)*

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[15]* for more information about the service `Det_ReportError`.

[SWS_BSW_00243][BSW Modules with enabled *development error* detection shall raise the *development error* `<MIP>_E_UNINIT` when any function apart from the `<Mip>_GetVersionInfo`, the `<Mip>_Init`, and the schedule functions [see SWS_BSW_00037] are called for an un-initialized BSW Module.] ()

7.2.3 Runtime errors

7.2.3.1 Synopsis

Runtime errors are specified as systematic faults that do not necessarily affect the overall system behavior.

For instance, wrong post-build configurations or wrongly assigned PDU-IDs are typical causes for runtime errors.

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 SRS_BSW_00452.

[SWS_BSW_00219] Runtime error type
⌈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` [ECUC_Det_00010].

7.2.3.4 Reporting

Any detected *runtime error* shall be reported:

[SWS_BSW_00222] Report detected *Runtime errors* to *Det*
⌈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*[15] for more information about the service `Det_ReportRuntimeError`.

7.2.4 Transient faults

7.2.4.1 Synopsis

Transient faults are caused by dysfunctional hardware. They occur if thermal noise or particle radiation influences the functionality of the hardware and so the functionality of the software connected with it. That also means that transient errors may heal, because the cause for the fault may disappear, again.

For instance, a CAN controller could go off-line due to a bit-flip in its control registers, induced by particle radiation.

Transient faults are reported to the module *Det* through the interface `Det_ReportTransientFault`. Although a certain implementation is not stipulated, *SRS_BSW_00473* requires that transient faults will not cause to stop the control flow of execution of the software.

The handling of those *transient faults* may require use case dependent actions.

Therefore, it is most likely that particular error handlers are implemented as callouts by an integrator. In this case the service `Det_ReportTransientFault` returns the return value of the callout function, otherwise it returns immediately with `E_OK`.

Monitors dedicated to detect *transient faults* must stay in the deployment build (production code).

7.2.4.2 Documentation

The SWS shall list the transient faults in its chapter 7 in accordance with the classification of SRS_BSW_00473.

[SWS_BSW_00223]Transient faults type
⌈Transient faults values are of Type *uint8.* ⌋*()*

7.2.4.3 Configuration

[SWS_BSW_00224] Detection of transient faults
⌈The detection of transient faults cannot be switched off, unless the Module SWS describes configuration parameters or other conditions, which define the activation of certain transient faults. ⌋*()*

If the *Det* implements the handling of transient faults by a callout function, then the particular callout function name of the Det must be configured by `DetReportTransientFaultCallout`[ECUC_Det_00011].

7.2.4.4 Reporting

[SWS_BSW_00225]Report detected Transient faults to Det
⌈The BSW Module shall report detected transient faults to the Default error tracer (*Det*) using the service `Det_ReportTransientFaults`.⌋()

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

See chapter 7.2.4 “Transient faults” activation and deactivation of *Development error* detection. See the Specification of *Det*[15] for more information about the service `Det_ReportRuntimeError`.

7.2.5 Extended production errors and production errors

7.2.5.1 Synopsis Production errors

According to SRS_BSW_00458 *production errors* are caused by any hardware problems, e.g., aging, deterioration, total hardware failure, bad production quality, incorrect assembly, etc. These hardware problems qualify for being production errors, if at least one of the following criteria is met (cf. 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, SRS_BSW_00458 and 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.

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.

7.2.5.2 Synopsis Extended production errors

Extended production errors indicate, like production errors, hardware problems or misbehavior of the environment (cf. SRS_BSW_00466).

Unlike production errors, however, extended production errors are not “first-class citizens” which means either that they do not meet any criteria of SRS_BSW_00458 or that the error points to the same root cause as an already defined production error [SRS_BSW_00472].

In this spirit, extended *production errors* may be utilized:

- to gain more information about the real cause of a corresponding production error
- to come to “deduced entries into the event memories” as a result of the combination of various information representing a certain ECU state

Extended production errors are also reported to the module *Dem*.

However, the appearance of a 'fail' state of a specific extended *production error* must not lead to an immediate entry into the primary event memory. Thus, extended *production errors* may utilze all features of the *Dem*, except the one to bind an error to an entry of the primary event memory directly.

It may be good practice to attach extended production errors to callback routines. It is then the responsibility of an ECU integrator to provide reasonable implementations. In this respect, the integrator still has every freedom, even to trigger an entry into the primary event memory.

7.2.5.3 Documentation

[SWS_BSW_00204] Documentation of (extended) production errors
[For each *production error* and *extended production error*,
appropriate documentation shall be provided according to the AUTOSAR SWS template.]*()*

7.2.5.4 Configuration

[SWS_BSW_00205] Detection of (extended) production errors
[The detection of *production code errors* and extended *production errors* cannot be switched off, unless the Module SWS describes configuration parameters or other conditions, which define the activation of certain (extended) *production errors*.]*()*

7.2.5.5 Reporting

Event IDs of (extended) 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 SericeNeeds of the *BSW module* (through the *Dem* ECUC) prefixed with `DemConf_DemEventParameter_`
See ecuc_sws_2108 (AUTOSAR_TPS_ECUConfiguration.pdf "**3.4.5.2 Representation of Symbolic Names**").

[SWS_BSW_00143] Values for Event IDs of production errors and extended production errors are imported
[Values for *Event IDs* of (extended) *production errors* are assigned externally by the configuration of the *Dem* module.]*(SRS_BSW_00409)*

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

[SWS_BSW_00046] Report production errors and extended production errors to Dem
[The *BSW Module* shall report all detected *production errors* and *extended production errors* to the *Diagnostic Event Manager* (*Dem*) using the service `Dem_SetEventStatus` if this specific *production error* or *extended production error* has been configured for this *BSW Module*.](SRS_BSW_00339)

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

[SWS_BSW_00066] Report EventStatus to *Dem*
[For reporting an (extended) 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.
](*SRS_BSW_00339*)

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 predefined generic signal de-bouncing algorithms.

[Note]
The callback service <Mip>_InitMonitorForEvent<EventName> is principally specified by the specification [Dem256] within Section 8.4.3.1.1 of the specification document for the module *Diagnostic Event Manager* (*Dem*). This document only specifies extensions which matter for the correct functionality of error monitors.

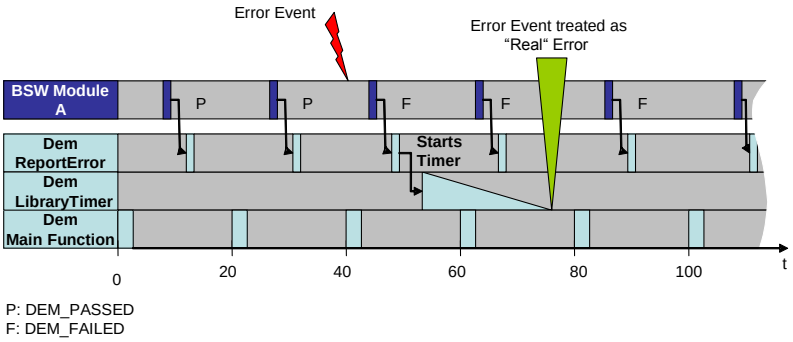
[SWS_BSW_00206] Only event-based error monitors shall implement the callback service
[<Mip>_InitMonitorForEvent<EventName>.

[Note]
The BSW module Dem calls an implemented callback service <Mip>_InitMonitorForEvent<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] On each trigger of the callback service [<Mip>_InitMonitorForEvent<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] If a particular BSW module implements a callback service [<Mip>_InitMonitorForEvent<EventName>, then the BSWMD shall specify a corresponding ServiceNeeds.]()

7.2.5.6 Example use case: Error is detected and notified



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

7.2.6 Security events

7.2.6.1 Synopsis

According to SRS_Crypto Stack_00122 security events are triggered by security related BSW modules or SWCs and shall be recorded in a security event memory (SEM).
These events qualify for being 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 in a SEM 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 captured in a SEM for later analysis. An example for such an event is the failed MAC verification of a secured PDU.

A particular *security event* 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 user defined memory dedicated to the SEM. A 'pass' may not revoke this entry.

.

7.2.6.2 Documentation

[SWS_BSW_00244] Documentation of security events
⌈For each *security event*, appropriate documentation shall be provided according to the AUTOSAR SWS template. ⌋*()*

7.2.6.3 Configuration

[SWS_BSW_00245] Detection of security events
⌈ The detection of security events is enabled by default, unless the SWS module describes configuration parameters or other conditions that define the (de)activation of certain security events. ⌋*()*

7.2.6.4 Reporting

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

The `EventId` symbols of security events are the short name of the ServiceNeeds of the *BSW module* (through the *Dem* ECUC) prefixed with `DemConf_DemEventParameter_`
See **TPS_ECUC_02108** (AUTOSAR_TPS_ECUConfiguration.pdf "**2.4.5.2 Representation of Symbolic Names**").

[SWS_BSW_00246] Values for Event IDs of security events are imported
⌈Values for *Event IDs* of security events are assigned externally by the configuration of the *Dem* module.⌋*(SRS_BSW_00409)*

For reporting *security events*, the *Dem* interface `Dem_SetEventStatusWithMonitorData` is used:

[SWS_BSW_00247] Report security events to Dem
⌈The *BSW Module* shall report all detected *security events* to the *Diagnostic Event Manager* (*Dem*) using the service `Dem_SetEventStatusWithMonitorData` if this specific *security event* has been configured for this *BSW Module*.⌋(SRS_BSW_00339)

Note that the configuration of *security events* is optional in the ECU Configuration of the *BSW Modules*.

[SWS_BSW_00248] Report EventStatus to *Dem*
⌈For reporting a security event state the following BSW specific interface of DEM shall be called:

```
Std_ReturnType Dem_SetEventStatusWithMonitorData (
    Dem_EventIdType EventId,
    Dem_EventStatusType EventStatus,
    Dem_MonitorDataType monitorData0,
```

```
        Dem_MonitorDataType monitorData1
    )
```

If a security event occurred `EventStatus` shall be equal to:
`'DEM_EVENT_STATUS_FAILED'.`
If a security event is not detected with sufficient precision and requires maturing by pre-debouncing `EventStatus` shall be equal to:
`'DEM_EVENT_STATUS_PREFAILED'.`
If a check is not possible (e.g., requires specific operating mode), no result shall be reported.
]*(SRS_BSW_00339)*

Note: the return value of Dem_SetEventStatusWithMonitorData 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.

7.2.7 Specific topics

7.2.7.1 Implementation specific errors

[SWS_BSW_00073] Implementation specific errors
[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.] *(SRS_BSW_00337)*

7.2.7.2 Handling of Symbolic Name Values

[SWS_BSW_00200] Symbolic Name values
[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*:

1. 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);
```

2. 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);
```

7.3 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

[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*

[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_00241] Alignment of *Meta Data*

⌈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*

⌈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.
…	…	…

8.2 Type definitions

[SWS_BSW_00146] Naming conventions for data types
[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*.]*(SRS_BSW_00305)*

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][

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*
[Data types used in Standardized Interface and Standardized AUTOSAR Interface shall only be defined in *RTE Types* header file (`Rte_Type.h`).]*(SRS_BSW_00447)*

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 (Synchronous / Asynchronous)	
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.	

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.

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

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
[All input parameters which are passed as pointers shall use the type qualifier “const”. The compiler abstraction macro P2CONST must be use.]()

For example:
```
Std_ReturnType <Mip>_DoWithInputBuffer (void* Buffer)
Shall be changed to

Std_ReturnType <Mip>_DoWithInputBuffer (
       P2CONST(void,AUTOMATIC,<MIP>_APPL_DATA) )
```

[SWS_BSW_00187] Input-Output Pointer parameters
[All INOUT / OUT parameters which are passed as pointers shall use the compiler abstraction macro P2VAR.]()

For example:
```
Std_ReturnType <Mip>_DoWithInOutBuffer (uint8* Buffer)

Shall be changed to

Std_ReturnType <Mip>_DoWithInOutBuffer (
       P2VAR(uint8,AUTOMATIC,<MIP>_APPL_DATA) )
```

[SWS_BSW_00049] Implement API parameter checking
[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.]*(SRS_BSW_00323, SRS_BSW_00414)*

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
[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 CopyRxData and CopyTxData with SduLength set to zero, or the RetryInfoPtr in CopyTxData if retry is not supported.

[SWS_BSW_00149] Do not pass function pointers as API parameter
[Function pointers shall not be passed as API parameter.]*(SRS_BSW_00371)*

The SWS_BSW_00149 just satisfies the negative requirement *SRS_BSW_00371*.

If different instances of the *BSW Module* are used, it may be necessary to differentiate API calls through an instance index.

[SWS_BSW_00047] Implement index based API services
[If different instances of the *BSW Module* are characterized by:

- same vendor and
- same functionality and
- same hardware device

then their API shall be accessed index based.]*(SRS_BSW_00413)*

Example:

```
MyFunction(uint8 MyIdx, MyType MyParameters, ... );
```

Or, optimized for source-code delivery:

```
#define MyInstance(index, p) Function##index (p)
```

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 *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
[Only the *ECU State Manager* and *Basic Software Mode Manager* are allowed to call *Initialization functions*.](*SRS_BSW_00101, SRS_BSW_00467*)

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_00058.

[SWS_BSW_00050] Check parameters passed to *Initialization functions*
[If the parameter checking for the *Initialization function* is enabled (SWS_BSW_00049), the *Configuration pointer* argument shall be checked with the following conditions:

- In the supportedConfigVariants VariantPreCompile and VariantLinkTime if only one configuration variant set is used, the initialization function does not need nor evaluate the passed argument. Thus the *Configuration pointer* shall have a `NULL_PTR` value.
- In the supportedConfigVariant VariantPostBuild or if multiple configuration variant sets are used, the initialization function requires the passed argument. Thus the *Configuration pointer* shall be different from `NULL_PTR`.

If these conditions are not satisfied, a *Development error* with type "Invalid configuration set selection" shall be reported to *Default Error Tracer* (*Det)*.
](*SRS_BSW_00414, SRS_BSW_00400, SRS_BSW_00438*)

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
[The state of a *BSW Module* shall be set accordingly at the end of *Initialization function*.](*SRS_BSW_00450*)
Note: This is used for *Development errors* detection

[SWS_BSW_00230] Call to *Initialization functions*
[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 or `<Mip>_GetVerisonInfo()` is always possible.]()

[SWS_BSW_00231] Multiple calls to *Initialization functions*
[The module *initialization function* shall not be called more than one time. 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 function*. This section contains general requirements valid for all module specific implementations of a *De-initialization function* service.

The *De-initialization function* API name follows *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
[Only the ECU State Manager and Basic Software Mode Manager are allowed to call *De-Initialization functions*.](*SRS_BSW_00467*)

[SWS_BSW_00072] Module state after *De-Initialization function*
[The state of a *BSW Module* shall be set accordingly at the beginning of the *De-Initialization function*.](*SRS_BSW_00450*)
Note: This is used for Development errors detection

[SWS_BSW_00232] Call to De-Initialization functions
[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
[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 Get Version Information

This section contains general requirements valid for all module specific implementations of the *Get Version Information* service.

[SWS_BSW_00064] Execution behavior of *Get Version Information*
[*Get Version Information* function shall be executed synchronously to its call and shall be reentrant.]*(SRS_BSW_00407)*

[SWS_BSW_00052] Return result from *Get Version Information*
[*Get Version Information* function shall have only one parameter. This parameter shall return the version information of this *BSW Module* with type `Std_VersionInfoType`, imported from *Standard Types* header (`Std_Types.h`).]*(SRS_BSW_00407)*

Note that the parameter name is part of each BSW Module specification.

The returned version information has type `Std_VersionInfoType`, which includes *Published information* from this module (see also SWS_BSW_00059 and AUTOSAR Specification of Standard Types [12]):

- Vendor Id
- Module Id
- Vendor specific version number

[SWS_BSW_00051] Configuration parameter for enabling *Get Version Information* service
[The availability of the *Get Version Information API* is configurable at *Pre-compile time* for every single *BSW Module*. The configuration parameter name shall be formed in the following way:

```
<Ma>VersionInfoApi
```

]*(SRS_BSW_00411)*

Example:

```
/* File: Eep_21_LDExt_Cfg.h
      */
#define EEP_21_LDEXT_VERSION_INFO_API STD_ON /*API is
enabled */
```

Note that for switching this configuration, the standard types `STD_ON` and `STD_OFF` shall be used (SWS_BSW_00029).

Note that if source code for both caller and callee of *Get Version Information* service *are* available, the *Implementation source* of the *BSW Module* may realize `<Mip>_GetVersionInfo` as a macro, defined in its *Implementation header* file.

Note: If `<Mip>_GetVersionInfo` is provided as a macro and a function is required, the provided macro could additionally be wrapped by a function definition.

[SWS_BSW_00236] Default configuration value of *Get Version Information*
[The availability of an *API* to *Get Version Information* from a *BSW Module* shall be configurable and the default value of the configuration shall be that this *API* is not available.**]** **()**

[SWS_BSW_00164] No restriction to Get Version Information calling context
⌈It shall be possible to call *Get Version Information* function at any time (e.g. before the *Initialization function* is called). ⌋*(SRS_BSW_00407)*

API configuration:

- The configuration of *Published information* (SWS_BSW_00059) of this *BSW Module* affects the API return values.

Please refer to the according *BSW Module* specification for further implementation details.

8.4 Callback notifications

Callbacks are functions, which are used for notifications to other modules.

Callbacks, which are AUTOSAR Services, follow the signature expected by the RTE. In this case, the return value of these functions has the type `Std_ReturnType` and the caller can assume, that always `E_OK` is returned. *Callback functions* should never fail, but this can happen, e.g. in partitioned systems

[SWS_BSW_00180] Signature of *Callback functions* of *AUTOSAR Services*
⌈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.⌋*(SRS_BSW_00440)*

[SWS_BSW_00172] Avoid return types other than `void` in *Callback functions*
⌈If the *BSW Module* provides *Callback functions* which are not *AUTOSAR Services*, then the return type of these functions shall avoid types other than `void`.⌋*(SRS_BSW_00359)*

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 function* parameters
⌈*Callback functions* are allowed to have parameters.⌋*(SRS_BSW_00360)*

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

8.5 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

[SWS_BSW_00153] Naming convention for *Scheduled functions*
[*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.]*(SRS_BSW_00373, SRS_BSW_00347)*

Examples (for illustration only):

a) Possible main processing function of *EEPROM* driver:
```
void Eep_21_LDExt_MainFunction(void)
```

b) 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
[*Scheduled functions* shall have no parameters and no return value. Their return type is always `void`.]*(SRS_BSW_00373)*

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
[*Scheduled functions* shall not enter any wait state.]*(SRS_BSW_00424)*

Typically, basic tasks are more efficient then extended tasks. Extended and basic task are classified in the *Specification of Operating System* [8].

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

8.6 Expected Interfaces

8.6.1 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
…	…	...

8.6.2 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
…	…	...

8.6.3 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 (Synchronous / Asynchronous)	
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.	

8.7 Service Interfaces

[SWS_BSW_00238] *ModeDeclarationGroups* definition in *BSWMD*
[*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*] (*SRS_BSW_00334*)

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 [2]
- AUTOSAR ECU Configuration Specification
- 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*
[Different use cases require different kinds of configurability. Therefore, the following *configuration variants* are provided:

- VARIANT-PRE-COMPILE: Allows individual configuration parameters to be realized at *"Pre-compile time"* only.
- VARIANT-LINK-TIME: Allows individual configuration parameters to be realized at either *"Pre-compile time"* or *"Link time"*.
- VARIANT-POST-BUILD: Allows individual configuration parameters to be realized at either *"Pre-compile time", "Link time"* or *"Post-build 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 Variant Multiplicity	• ***True****: This container may have different number of instances in different post-build variants (previously known as post-build selectable configuration sets).* • ***False****: This container may NOT have different number of instances in different post-build variants (previously known as post-build selectable configuration sets).*
Configuration Parameters	

10.1.4.2 Configuration parameter section:

Name	*Identifies the parameter by name.*	
Description	*Explains the intention of the configuration parameter.*	
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 Variant Multiplicity	• ***True***: *This parameter may have different number of instances in different post-build variants (previously known as post-build selectable configuration sets).* • ***False***: *This parameter may NOT have different number of instances in different post-build variants (previously known as post-build selectable configuration sets).*		
Post-Build Variant Value	• ***True***: *This parameter may have different value in different post-build variants (previously known as post-build selectable configuration sets).* • ***False***: *This parameter may NOT have different value in different post-build variants (previously known as post-build selectable configuration sets).*		
Multiplicity Configuration Class	***Pre-compile***	*see*[1]	*Reference to (a) variant(s).*
	Link time	*see*[2]	*Reference to (a) variant(s).*
	Post Build	*see*[3]	*Reference to (a) variant(s).*
Value Configuration Class	***Pre-compile***	*see*[4]	*Reference to (a) variant(s).*
	Link time	*see*[5]	*Reference to (a) variant(s).*
	Post Build	*see*[6]	*Reference to (a) variant(s).*
Scope	• ***LOCAL*** *: The parameter is applicable only for the module it is defined in* • ***ECU*** *: The parameter may be shared with other modules (i.e. exported)*		
Dependency	*Describe the dependencies with respect to the scope if known ot mark it as "- -".*		

10.1.4.3 Section of included/referenced containers:

Included Containers		
Container Name	***Multiplicity***	***Scope / 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 scope of the referenced sub-container if known or mark it as "- -".* *The scope describes the impact of the configuration parameter: Does the setting affect only one instance of the module (instance), all instances of this module (module), the ECU or a network.* *Possible values of scope : instance, module, ECU, network>* *Describes the dependencies with respect to the scope if known or mark it as "- -".*

[1] see the explanation for configuration class label: Pre-compile time
[2] see the explanation for configuration class label: Link time
[3] see the explanation for configuration class label: Post Build time
[4] see the explanation for configuration class label: Pre-compile time
[5] see the explanation for configuration class label: Link time
[6] see the explanation for configuration class label: Post Build time

10.1.5 Configuration class labels

The configuration parameter section is complemented by a label with additional specification for each type of configuration class:

Pre-compile time: Specifies whether the configuration parameter shall be of configuration class *Pre-compile time* or not.

Label	*Description*
x	The configuration parameter shall be of configuration class *Pre-compile time*.
--	The configuration parameter shall never be of configuration class *Pre-compile time*.

Link time: Specifies whether the configuration parameter shall be of configuration class *Link time* or not.

Label	*Description*
x	The configuration parameter shall be of configuration class *Link time*.
--	The configuration parameter shall never be of configuration class *Link time*.

Post Build: Specifies whether the configuration parameter shall be of configuration class *Post Build* or not.

Label	*Description*
x	The configuration parameter shall be of configuration class *Post Build* and no specific implementation is required.
--	The configuration parameter shall never be of configuration class *Post Build*.

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
[Files holding configuration data for the *BSW Module* shall have a format that is readable and understandable by human beings.]*(SRS_BSW_00160)*

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.

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
[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](SRS_BSW_00397).

All *Pre-compile time configuration* parameters are defined in the *Pre-compile time configuration source* (SWS_BSW_00012) or in the *Pre-compile time configuration header* (SWS_BSW_00031).

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
[The configuration of *BSW Modules* with link time parameters is achieved on object code basis in the stage after compiling and before linking] (SRS_BSW_00398). *See also* [SWS_BSW_00117].

[SWS_BSW_00056] Configuration pointer to link-time configurable data
[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.]*(SRS_BSW_00344)*

All *Link time configuration* parameters are defined in the *Link time configuration source* (SWS_BSW_00014) and declared in the *Link time configuration header* (SWS_BSW_00033).

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*
[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*.]*(SRS_BSW_00438)*

[SWS_BSW_00158] Use of *Configuration pointers* to *Post-build configuration data structure* is restricted
[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*.]*(SRS_BSW_00438)*

Post-build configuration data is located in a separate segment and can be loaded independently of the actual code [7]. 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
[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.]*(SRS_BSW_00404)*

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.

All *Post-build time configuration* parameters are defined in the *Post-build time configuration source* (SWS_BSW_00015) and declared in the *Post-build time configuration header* (SWS_BSW_00035).

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
⌈ 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
⌈ 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
⌈ 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.⌋ (SRS_BSW_00400, SRS_BSW_00405)

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*
[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:

Published information elements		
Information element	***Type / Range***	***Information element description***

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 the BSW Module List [1].
<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.

The *Published information* is configured in the *BSW Module Description* [4] for this module.]*(SRS_BSW_00402, SRS_BSW_00003, SRS_BSW_00379, SRS_BSW_00374, SRS_BSW_00318, SRS_BSW_00407)*

[SWS_BSW_00161] Restriction to declaration of vendor identification
[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].]*(SRS_BSW_00374)*

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 BSW Modules List [1].

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
[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.]*(SRS_BSW_00321)*

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