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

Document Version	1.2.0
Document Status	Final
Part of Release	4.1
Revision	3

Document Change History

Date	Version	Changed by	Change Description
31.03.2014	1.2.0	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
31.10.2013	1.1.0	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
05.03.2013	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

Advice for users

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

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

Table of 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 [14], 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*_00001,* SRS_BSW_0000*2)*

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.

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 Debugging
AUTOSAR_SWS_Debugging.pdf

[13] AUTOSAR Specification of Standard Types
AUTOSAR_SWS_PlatformTypes.pdf

[14] AUTOSAR Standardization Template
AUTOSAR_TPS_StandardizationTemplate.pdf

[15] AUTOSAR Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[16] AUTOSAR Specification of Development Error Tracer
AUTOSAR_SWS_DevelopmentErrorTracer.pdf

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

3.2 Related standards and norms

[18] MISRA C 2004 Standard
Homepage: http://www.misra.org.uk/

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

[20] HIS Software Supplier Identifications
http://www.automotive--his.de/his--ergebnisse.htm

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*.]*(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 *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_LDExtEepDriver`: Prefix for EEPROM driver implementation, where `vendorApiInfix` and `vendorId` are identified by “LDExtEepDriver” and “21” respectively.

Examples of *Module abbreviations*:

- `FrIf`: FlexRay Inteface 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`
Callback header	conditional	`Com_Cbk.h`
Debugging header	conditional	`Com_Dbg.h`
Pre-compile time configuration source	conditional	`Com_Cfg.c`
Pre-compile time configuration header	conditional	`Com_Cfg.h`
Link time configuration source	conditional	`Com_Lcfg.c`
Link time configuration header	conditional	`Com_Lcfg.h`
Post-build time configuration source	conditional	`Com_PBcfg.c`
Post-build time configuration header	conditional	`Com_PBcfg.h`
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 toolchain
[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
- Toolchain name and version
- Toolchain 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_00436)*

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

Example:
The CAN Driver *Module implementation* file `Can.c` includes the header file `SchM_Can.h`.

[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_00435)*

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].

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 also chapter 7.2 – Error .

[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_00370)*

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_00370)

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 Pre-compile time configuration source

The *Pre-compile time configuration source* contains definitions of pre-compile time configuration parameters for the *BSW Module*. More specifically, for those parameters that are defined as `const`.

[SWS_BSW_00011] Provide *Pre-compile time configuration source* files
[If the *BSW Module* implementation contains pre-compile time configuration parameters defined as `const`, then the code file structure shall contain one or more files for their definition: the *Pre-compile time configuration source* files. The file names shall be formed in the following way:

```
<Mip>[_<Ie>]_Cfg.c
```

]*(SRS_BSW_00346, SRS_BSW_00345, SRS_BSW_00419)*

[SWS_BSW_00012] Define all *Pre-compile time configuration parameters* (`const`)
[The *Pre-compile time configuration source* shall contain definitions for all pre-compile time configuration parameters that are defined as `const` and are specified for this module.]*(SRS_BSW_00158, SRS_BSW_00345, SRS_BSW_00419)*

See also chapter 10.2.3 - Pre-compile time configuration.

5.1.6.3 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, 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
```

]*(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.

5.1.6.4 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* shall contain definitions for all post-build time configuration parameters specified for this module.]*(SRS_BSW_00158, SRS_BSW_00380)*

See also chapter 10.2.5 - Post-build time configuration.

5.1.6.5 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 7.1.19 - 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.

5.1.7.3 Callback header

[SWS_BSW_00026] Provide *Callback header* files
[If the *BSW Module* implementation contains *Callback functions*, then the header file structure shall contain one or more files that provide their declarations: the *Callback header* files. The file names shall be formed in the following way:

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

]*(SRS_BSW_00346, SRS_BSW_00370)*

Example:
The *Callback header* content for module NVRAM Manager may look like this:

```
/* File: NvM_Cbk.h                    */
```

```
…
void NvM_NotifyJobOk ( void );
void NvM_NotifyJobError (void );
…
```

The separation of callback declaration from explicitly exported module functions is necessary to prevent misuse of unintentionally exposed API. Only modules calling callbacks of this module need to include its *Callback header*.

Please refer to chapter 8.4 “Callback notifications” of according *BSW Module* SWS for information on callbacks defined for this module.

5.1.7.4 Debugging header

The implementation of debugging features is optional in AUTOSAR modules. The debugging support is helped by the definition of *Debugging variables*.

[SWS_BSW_00027] Provide *Debugging header* files
[If the *BSW Module* implementation contains *Debugging variables* the header file structure shall contain one or more files that provide their declarations: the *Debugging header* files. The file name shall be formed in the following way:

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

]*(SRS_BSW_00346, SRS_BSW_00442)*

[SWS_BSW_00028] Declare *Debugging variables*
[If the *BSW Module* implementation contains *Debugging variables*, their declaration shall be provided in its *Debugging header* file or files.]*(SRS_BSW_00442)*

[SWS_BSW_00141] Access to type definitions of *Debugging variables*
[If the *BSW Module* implementation contains *Debugging variables*, all type definitions of *Debugging variables* shall be accessible by including the module *Implementation header*.]*(SRS_BSW_00442)*

For further information, see chapter 7.1.18 - Debugging support.

5.1.7.5 Pre-compile time configuration header

The Pre-compile time configuration header contains definitions of pre-compile time configuration parameters for the BSW Module.

[**SWS_BSW_00030]** Provide *Pre-compile time configuration header* files
[All *BSW Module* implementation contains definitions of pre-compile time configuration parameters which are defined as pre-processor directives (`#define`). The code file structure shall contain one or more files for the definition of these parameters: the *Pre-compile time configuration header* files. The file names shall be formed in the following way:

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

](SRS_BSW_00346, SRS_BSW_00381)

[SWS_BSW_00031] Define all *Pre-compile time configuration parameters* (`#define`)
[The *Pre-compile time configuration header* shall contain definitions for all *Pre-compile time configuration parameters* defined as pre-processor directive (`#define`) which are specified for this *BSW Module*.](SRS_BSW_00158, SRS_BSW_00345, SRS_BSW_00381)

Example:
The pre-processor switches for *Eep* module are defined in `Eep_21_LDExtEepDriver_Cfg.h`.

See also chapter 10.2.3 - Pre-compile time configuration.

5.1.7.6 Link time configuration header

The Link time configuration header contains declarations of link time configuration parameters for this BSW Module.

[SWS_BSW_00032] Provide *Link time configuration header* files
[If the *BSW Module* implementation contains link time configuration parameters, the code file structure shall contain one or more files for their declaration: the *Link time configuration header* files. The file names shall be formed in the following way:

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

](SRS_BSW_00346)

[SWS_BSW_00033] Declare all *Link time configuration parameters*
[The *Link time configuration header* files shall contain declarations for all link time configuration parameters specified for this *BSW Module*.](SRS_BSW_00158, SRS_BSW_00380)

See also chapter 10.2.4 - Link time configuration.

5.1.7.7 Post-build time configuration header

The Post-build time configuration header contains declarations of post-build time configuration parameters for the BSW Module.

[**SWS_BSW_00034]** Provide *Post-build time configuration header* files
[If the *BSW Module* implementation contains post-build time configuration parameters, the code file structure shall contain one or more files for declaration of these parameters: the *Post-build time configuration header* files. The file names shall be formed in the following way:

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

]*(SRS_BSW_00346)*

[SWS_BSW_00035] Declare all *Post-build time configuration parameters*
[The *Post-build time configuration header* files shall contain declarations for all post-build time configuration parameters specified for this *BSW Module*.]*(SRS_BSW_00158, SRS_BSW_00380)*

See also chapter 10.2.5 - Post-build time configuration.

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]

[SRS_BSW_00344] Reference to link-time configuration	SWS_BSW_00056
[SRS_BSW_00404] Reference to post build time configuration	SWS_BSW_00160
[SRS_BSW_00405] Reference to multiple configuration sets	SWS_BSW_00160
[SRS_BSW_00345] Pre-compile time configuration	SWS_BSW_00011, SWS_BSW_00012, SWS_BSW_00031
[SRS_BSW_00159] Tool-based configuration	SWS_BSW_00116
[SRS_BSW_00167] Static configuration checking	SWS_BSW_00061

[SRS_BSW_00171] Configurability of optional functionality	SWS_BSW_00029
[SRS_BSW_00170] Data for reconfiguration of SW-components	SWS_BSW_SPEC
[SRS_BSW_00380] Separate C-Files for configuration parameters	SWS_BSW_00014, SWS_BSW_00063, SWS_BSW_00033, SWS_BSW_00035
[SRS_BSW_00419] Separate C-Files for pre-compile time configuration parameters	SWS_BSW_00011, SWS_BSW_00012
[SRS_BSW_00381] Separate configuration header file for pre-compile time parameters	SWS_BSW_00030, SWS_BSW_00031
[SRS_BSW_00412] Separate H-File for configuration parameters	Clarification necessary.
[SRS_BSW_00383] List dependencies of configuration files	SWS_BSW_SPEC
[SRS_BSW_00384] List dependencies to other modules	SWS_BSW_SPEC
[SRS_BSW_00387] Specify the configuration class of callback function	SWS_BSW_SPEC
[SRS_BSW_00388] Introduce containers	SWS_BSW_SPEC
[SRS_BSW_00389] Containers shall have names	SWS_BSW_SPEC
[SRS_BSW_00390] Parameter content shall be unique within the module	SWS_BSW_SPEC
[SRS_BSW_00391] Parameter shall have unique names	SWS_BSW_SPEC
[SRS_BSW_00392] Parameters shall have a type	SWS_BSW_SPEC
[SRS_BSW_00393] Parameters shall have a range	SWS_BSW_SPEC
[SRS_BSW_00394] Specify the scope of the parameters	SWS_BSW_SPEC
[SRS_BSW_00395] List the required parameters (per parameter)	SWS_BSW_SPEC
[SRS_BSW_00396] Configuration classes	SWS_BSW_SPEC
[SRS_BSW_00397] Pre-compile-time parameters	SWS_BSW_00183
[SRS_BSW_00398] Link-time parameters	SWS_BSW_00184
[SRS_BSW_00399] Loadable Post-build time parameters	SWS_BSW_00159
[SRS_BSW_00400] Selectable Post-build time parameters	SWS_BSW_00050, SWS_BSW_00058
[SRS_BSW_00438] Post Build Configuration Data Structure	SWS_BSW_00057, SWS_BSW_00158, SWS_BSW_00050
[SRS_BSW_00402] Published information	SWS_BSW_00059
[SRS_BSW_00375] Notification of wake-up reason	SWS_BSW_SPEC
[SRS_BSW_00101] Initialization interface	SWS_BSW_00150, SWS_BSW_SPEC
[SRS_BSW_00416] Sequence of Initialization	SWS_BSW_SPEC
[SRS_BSW_00406] Check module initialization	SWS_BSW_SPEC
[SRS_BSW_00467] Calling of init / deinit	SWS_BSW_00150, SWS_BSW_00152
[SRS_BSW_00437] NoInit--Area in RAM	SWS_BSW_SPEC
[SRS_BSW_00168] Diagnostic Interface of SW components	SWS_BSW_SPEC
[SRS_BSW_00407] Function to read out published parameters	SWS_BSW_00052, SWS_BSW_00164, SWS_BSW_00059, SWS_BSW_00064, SWS_BSW_00168,
[SRS_BSW_00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	SWS_BSW_00001, SWS_BSW_00040
[SRS_BSW_00424] BSW main processing function task allocation	SWS_BSW_00156
[SRS_BSW_00425] Trigger conditions for schedulable objects	SWS_BSW_SPEC

[SRS_BSW_00426] Exclusive areas in BSW modules	SWS_BSW_00001, SWS_BSW_00038, SWS_BSW_00134,
[SRS_BSW_00427] ISR description for BSW modules	SWS_BSW_00001, SWS_BSW_00041, SWS_BSW_00065, SWS_BSW_SPEC
[SRS_BSW_00428] Execution order dependencies of main processing functions	SWS_BSW_00054
[SRS_BSW_00429] Restricted BSW OS functionality access	SWS_BSW_00138
[SRS_BSW_00432] Modules should have separate main processing functions for read/receive and write/transmit data path	SWS_BSW_SPEC
[SRS_BSW_00433] Calling of main processing functions	SWS_BSW_00133
[SRS_BSW_00450] Main Function Processing for Un-Initialized Modules	SWS_BSW_00037; SWS_BSW_00071; SWS_BSW_00072;
[SRS_BSW_00442] Debugging Support in Modules	SWS_BSW_00027, SWS_BSW_00028, SWS_BSW_00141, SWS_BSW_00139, SWS_BSW_00140, SWS_BSW_00044, SWS_BSW_SPEC
[SRS_BSW_00461]	SWS_BSW_SPEC
[SRS_BSW_00336] Shutdown interface	SWS_BSW_SPEC
[SRS_BSW_00337] Classification of errors	SWS_BSW_00144, SWS_BSW_00042, SWS_BSW_00073
[SRS_BSW_00338] Detection and Reporting of development errors	SWS_BSW_00045, SWS_BSW_00042
[SRS_BSW_00369] Do not return development error codes via API	SWS_BSW_SPEC
[SRS_BSW_00339] Reporting of production relevant error status	SWS_BSW_00046, SWS_BSW_00066
[SRS_BSW_00422] Pre--de--bouncing of production relevant error status	SWS_BSW_00166
[SRS_BSW_00417] Reporting of Error Events by Non-Basic Software	SWS_BSW_SPEC
[SRS_BSW_00323] API parameter checking	SWS_BSW_00049 SWS_BSW_SPEC
[SRS_BSW_00004] Version check	SWS_BSW_00036
[SRS_BSW_00409] Header files for production code error IDs	SWS_BSW_00008, SWS_BSW_00143
[SRS_BSW_00385] List possible error notifications	SWS_BSW_SPEC
[SRS_BSW_00386] Configuration for detecting an error	SWS_BSW_SPEC
[SRS_BSW_00455]	SWS_BSW_SPEC
[SRS_BSW_00161] Microcontroller abstraction	SWS_BSW_SPEC
[SRS_BSW_00162] ECU layout abstraction	SWS_BSW_SPEC
[SRS_BSW_00005] No hard coded horizontal interfaces within MCAL	SWS_BSW_SPEC
[SRS_BSW_00415] User dependent include files	SWS_BSW_SPEC
[SRS_BSW_00164] Implementation of interrupt service routines	SWS_BSW_00137
[SRS_BSW_00325] Runtime of interrupt service routines	SWS_BSW_00167
[SRS_BSW_00326] Transition from ISRs to OS tasks	SWS_BSW_00182
[SRS_BSW_00342] Usage of source code and object code	SWS_BSW_00117
[SRS_BSW_00343] Specification and configuration of time	SWS_BSW_SPEC
[SRS_BSW_00160] Human-readable configuration data	SWS_BSW_00157

[SRS_BSW_00453] – Harmonization of BSW Modules	SWS_BSW_SPEC
[SRS_BSW_00456] - Header file for Harmonizing BSW Modules	SWS_BSW_SPEC
[SRS_BSW_00457] - Callback functions of Application software components	SWS_BSW_SPEC
[SRS_BSW_00007] HIS MISRA C	SWS_BSW_00115
[SRS_BSW_00300] Module naming convention	SWS_BSW_00101, SWS_BSW_00103
[SRS_BSW_00413] Accessing instances of BSW modules	SWS_BSW_00047, SWS_BSW_00148
[SRS_BSW_00347] Naming separation of different instances of BSW drivers	SWS_BSW_00102, SWS_BSW_00148, SWS_BSW_00153, SWS_BSW_00126, SWS_BSW_169
[SRS_BSW_00441] Enumeration literals and #define naming	SWS_BSW_00124, SWS_BSW_SPEC
[SRS_BSW_00305] Data types naming convention	SWS_BSW_00146, SWS_BSW_SPEC
[SRS_BSW_00307] Global variables naming convention	SWS_BSW_00130
[SRS_BSW_00310] API naming convention	SWS_BSW_00148, SWS_BSW_SPEC
[SRS_BSW_00373] Main processing function naming convention	SWS_BSW_00153, SWS_BSW_00154, SWS_BSW_SPEC
[SRS_BSW_00327] Error values naming convention	SWS_BSW_00125, SWS_BSW_SPEC
[SRS_BSW_00335] Status values naming convention	SWS_BSW_00124, SWS_BSW_SPEC
[SRS_BSW_00350] Development error detection keyword	SWS_BSW_00042
[SRS_BSW_00408] Configuration parameter naming convention	SWS_BSW_00126, SWS_BSW_SPEC
[SRS_BSW_00410] Compiler switches shall have defined values	SWS_BSW_00123
[SRS_BSW_00411] Get version info keyword	SWS_BSW_00051
[SRS_BSW_00463] Callout function prototype generation	SWS_BSW_00135, SWS_BSW_00136
[SRS_BSW_00464]File names´ case sensitivity	SWS_BSW_00170
[SRS_BSW_00465] Disambiguation rules on module names	SWS_BSW_00171
[SRS_BSW_00346] Basic set of module files	SWS_BSW_00004, SWS_BSW_00011, SWS_BSW_00013, SWS_BSW_00015, SWS_BSW_00020, SWS_BSW_00026, SWS_BSW_00027, SWS_BSW_00030, SWS_BSW_00032, SWS_BSW_00034
[SRS_BSW_00158] Separation of configuration from implementation	SWS_BSW_00012, SWS_BSW_00014, SWS_BSW_00063, SWS_BSW_00031, SWS_BSW_00033, SWS_BSW_00035
[SRS_BSW_00314] Separation of interrupt frames and service routines	SWS_BSW_00016, SWS_BSW_00017, SWS_BSW_00018, SWS_BSW_00019, SWS_BSW_00021, SWS_BSW_00066SWS_BSW_00181
[SRS_BSW_00370] Separation of callback interface from API	SWS_BSW_00009, SWS_BSW_00010, SWS_BSW_00026
[SRS_BSW_00435] Module Header File Structure for the Basic Software Scheduler	SWS_BSW_00007
[SRS_BSW_00436] Module Header File Structure for the Basic Software Memory Mapping	SWS_BSW_00006
[SRS_BSW_00447] Standardizing Include file structure of BSW Modules Implementing AUTOSAR Service	SWS_BSW_00023, SWS_BSW_025, SWS_BSW_00147, SWS_BSW_00069

[SRS_BSW_00348] Standard type header	SWS_BSW_00024
[SRS_BSW_00353] Platform specific type header	SWS_BSW_00120, SWS_BSW_00122
[SRS_BSW_00361] Compiler specific language extension header	SWS_BSW_00121
[SRS_BSW_00301] Limit imported information	SWS_BSW_00104, SWS_BSW_SPEC
[SRS_BSW_00302] Limit exported information	SWS_BSW_00105, SWS_BSW_SPEC
[SRS_BSW_00328] Avoid duplication of code	SWS_BSW_00127
[SRS_BSW_00312] Shared code shall be reentrant	SWS_BSW_SPEC
[SRS_BSW_00006] Platform independency	SWS_BSW_00119
[SRS_BSW_00439] Declaration of interrupt handlers and ISRs	SWS_BSW_00043, SWS_BSW_00068
[SRS_BSW_00448] Module SWS shall not contain requirements from Other Modules	SWS_BSW_SPEC
[SRS_BSW_00449] BSW Service APIs used by Autosar Application Software shall return a Std_ReturnType	SWS_BSW_SPEC
[SRS_BSW_00357] Standard API return type	SWS_BSW_SPEC
[SRS_BSW_00377] Module Specific API return type	SWS_BSW_SPEC
[SRS_BSW_00304] AUTOSAR integer data types	SWS_BSW_00120
[SRS_BSW_00355] Do not redefine AUTOSAR integer data types	SWS_BSW_00122,\
[SRS_BSW_00378] AUTOSAR Boolean type	SWS_BSW_00142
[SRS_BSW_00306] Avoid direct use of compiler and platform specific keywords	SWS_BSW_00121
[SRS_BSW_00308] Definition of global data	SWS_BSW_00129
[SRS_BSW_00309] Global data with read-only constraint	SWS_BSW_00131
[SRS_BSW_00371] Do not pass function pointers via API	SWS_BSW_00149
[SRS_BSW_00358] Return type of init() functions	SWS_BSW_00185
[SRS_BSW_00414] Parameter of init function	SWS_BSW_00049, SWS_BSW_00050, SWS_BSW_00151, SWS_BSW_SPEC
[SRS_BSW_00376] Return type and parameters of main processing functions	SWS_BSW_00154
[SRS_BSW_00359] Return type of callback functions	SWS_BSW_00172
[SRS_BSW_00360] Parameters of call-out functions	SWS_BSW_00173, SWS_BSW_SPEC
[SRS_BSW_00440] Function prototype for callback functions of AUTOSAR Services	SWS_BSW_00180
[SRS_BSW_00329] Avoidance of generic interfaces	SWS_BSW_SPEC
[SRS_BSW_00330] Usage of macros instead of functions	SWS_BSW_00132
[SRS_BSW_00331] Separation of error and status values	SWS_BSW_SPEC
[SRS_BSW_00462] Requirement Id for Standardized Autosar Interface	SWS_BSW_SPEC
[SRS_BSW_00009] Module User Documentation	SWS_BSW_00002
[SRS_BSW_00401] Documentation of multiple instances of configuration parameters	SWS_BSW_SPEC
[SRS_BSW_00172] Compatibility and documentation of scheduling strategy	SWS_BSW_SPEC
SRS_BSW_00010] Memory resource documentation	SWS_BSW_00002
[SRS_BSW_00333] Documentation of callback function context	SWS_BSW_00167, SWS_BSW_SPEC

[SRS_BSW_00374] Module vendor identification	SWS_BSW_00059, SWS_BSW_00161
[SRS_BSW_00379] Module identification	SWS_BSW_00059
[SRS_BSW_00003] Version identification	SWS_BSW_00059
[SRS_BSW_00318] Format of module version	SWS_BSW_00059
[SRS_BSW_00321] Enumeration of module version numbers	SWS_BSW_00162
[SRS_BSW_00341] Microcontroller compatibility documentation	SWS_BSW_00003
[SRS_BSW_00334] Provision of XML file	SWS_BSW_00001
[SRS_BSW_00451] Acces to HW registers	SWS_BSW_00179

7 Functional specification

7.1 General implementation specification

7.1.1 Conformance to MISRA C

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 2004 Standard [18].]*(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.

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 [15].

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 is 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 19.4, 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 according *BSW Module documentation*, which is delivered with the module implementation.

Optional functionalities of a *BSM Module* shall not consume resources (RAM, ROM and runtime). These functionalities are enabled or disabled at pre-compile time by the according 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_DEV_ERROR_DETECT == STD_ON )
…
```

Example of a wrong implementation:

```
#ifdef EEP_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_UNINIT
EEP_IDLE
EEP_BUSY
```

Examples for pre-processor directives:

```
#define EEP_PARAM_CONFIG
#define EEP_SIZE
```

Example for enumeration literals:

```
typedef enum
{
   EEP_DRA_CONFIG,
   EEP_ARE,
   EEP_EV
} Eep_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) `<SN>` 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_E_BUSY
EEP_E_PARAM_ADDRESS
EEP_E_PARAM_LENGTH
EEP_E_WRITE_FAILED
```

7.1.7 Configuration parameters

The *BSW Module* implementation must use *Configuration parameter names* and *Configuration parameter labels* derived from the respective configuration parameters specification. For further information, see also chapter 10.2.2- Implementation names.

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

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

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

Camel case:	`<Mip><Pn>`
Upper 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* and the respective *Configuration parameter label specification.*]*(SRS_BSW_00408, SRS_BSW_00347)*

Examples:

- `CanIfTxConfirmation`
- `PDUR_E_INIT_FAILED`
- `EEP_21_LDEXT_NORMAL_WRITE_BLOCK_SIZE`

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. Except *Debugging variables* – see chapter 7.1.18. "Debugging support" – the *BSW Module* shall not define global data in its header file.

[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. Exception: *Debugging Variables* are not affected by this rule, see SWS_BSW_00028.]*(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_MessageBuffer[CAN_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` modifier keyword.

[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` modifier to their definitions.]*(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.]*(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*.]*(BSW00460)*

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) <Mip>_<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 *Callback name*, which shall have the same spelling of the *Callback name*, including module reference, but written in upper case and `<Cn>` is the *Callback 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_COM_SOMECALLOUT_CODE
#include "Com_MemMap.h"
FUNC(void, COM_SOMECALLOUT_CODE) Com_SomeCallout(void);
#define COM_STOP_SEC_COM_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 according 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 memory protection
[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_00326)*

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	✓				
AllowAccess	✓				
GetApplicationState	✓				
ControlIdle	✓	✓			
GetNumberOfActivatedCores	✓				
GetCoreID	✓	✓	✓	✓	✓
StartCore		✓			

OS Services	RTE , BSW Scheduler, BswM, CDD	EcuM	MCAL	StbM	Other *BSW Modules*
StartNonAutosarCore		✓			
GetSpinlock	✓	✓	✓		
ReleaseSpinlock	✓	✓	✓		
TryToGetSpinlock	✓	✓	✓		
ShutdownAllCores		✓			

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] Concurent 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 Debugging support

The AUTOSAR architecture supports standardized debugging and tracing features for basic software, RTE and software components. The debugging feature is optional. Mainly, the debugging feature is supported by the definition of *Debugging variables* in the module implementation. See also the see chapter 3.1 [12].

Debugging variables are an individual implementer choice and cannot be standardized. If a *BSW Module* contains such variables, these variables have to be described in the *BSW Module Description*. According to this description, it is possible to derive their data size and data names and to configure the *Debugging* module (*Dbg*) [12].

[SWS_BSW_00044] Describe *Debugging variables*
[All *Debugging variables* shall be described within the respective *BSW Module Description* (SWS_BSW_00001) like specified in *BSW Module Description Template* [4].]*(SRS_BSW_00442)*

[SWS_BSW_00139] Debugging variables are global variables
[If the *BSW Module* supports debugging, each variable that shall be accessible for debugging (*Debugging variables*) shall be defined as global variable.]*(SRS_BSW_00442)*

[SWS_BSW_00140] Enable calculation of Debugging variables' size
[The declaration of *Debugging variables* shall be such, that it is possible to calculate the size of each variable by using the *C* operator `sizeof`.]*(SRS_BSW_00442)*

The declaration of *Debugging variables* is provided in the *Debugging header* of the *BSW Module*, see chapter 5.1.7.4 - Debugging header and requirements SWS_BSW_00027 and SWS_BSW_00028.

7.1.19 Data types

7.1.19.1 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.

7.1.19.2 Platform Specific Types

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

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

7.1.19.2.1 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.

[SWS_BSW_00122] Redefinition of integer data types is restricted
[The implementation shall not define own types on top of the AUTOSAR Integer Data Types if this is not necessary and the data size is known at specification time.]*(SRS_BSW_00355, SRS_BSW_00353)*

Example 1:
The data size of parameter `DeviceIndex` is known at specification time (8 bit). Hence the following is not allowed:

```
typedef uint8 DeviceIndexType  /* wrong! */
...
static DeviceIndexType DeviceIndex
```

Use the following instead:

```
static uint8 DeviceIndex
```

Example 2:
The parameter `DeviceAddress` is platform dependent (could by 16..32 bit). It is required for runtime efficiency, that the best type is chosen for a specific platform.

On 16 bit platforms:

```
typedef uint16 DeviceAddressType
```

On 32 bit platforms:

```
typedef uint32 DeviceAddressType
```

7.1.19.2.2 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 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.h: */
…
/* this automatically includes Platform_Types.h: */
#include "Std_Types.h"
…
boolean Eep_Busy(void) {…}
…


/* File: calling module */
…
if (Eep_Busy() == FALSE) {…}
…
```

7.1.20 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".](RS_BRF_01160)

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

7.2.1 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 providing 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_E_COM_FAILURE`) for the module *Eep*:

1. Example for **source code integration**:
```
/* File: Dem.h */
…
/* DEM specifies the production code error ID: */
#define DemConf_DemEventParameter_EEP_E_COM_FAILURE
((Dem_EventIdType) 14u)
…
/* File: Eep.c */
#include "Dem.h"
…
Dem_ReportErrorStatus(DemConf_DemEventParameter_EEP_E_COM_
FAILURE, DEM_EVENT_STATUS_PREFAILED);
```

2. Example for **object code integration**:
```
/* File: Dem.h */
```

```
…
/* DEM specifies the production code error ID: */
#define DemConf_DemEventParameter_EEP_E_COM_FAILURE
((Dem_EventIdType) 14u)
/* File: Eep_PBcfg.c
Post-build configuration source
This file needs to be compiled and linked with the
object code delivery: */
#include "Dem.h"
#include "Eep_cfg.h"
…
const Dem_EventIdType Eep_E_Com_Failure =
DemConf_DemEventParameter_EEP_E_COM_FAILURE;
…
/* File: Eep_cfg.h
This file needs to be compiled and linked with the
object code delivery: */
…
extern const Dem_EventIdType Eep_E_Com_Failure;
…
/* File: Eep.c
This file is delivered as object file. */
#include "Dem.h"
#include "Eep_cfg.h"
…
Dem_ReportErrorStatus( Eep_E_Com_Failure,
DEM_EVENT_STATUS_PREFAILED);
```

7.2.2 Error Classification

[SWS_BSW_00144] Error classification
[All errors, which may be detected and/or reported by the *BSW Module*, are classified in two different types:

- Development errors and integration errors are expected to occur only during development, and detect problems of the implementation or the integration. ()
- *Production errors and extended production errors which detect problems of the hardware, which may lead to fail-safe operation and/or can be repaired by the garage.* ()

]*(SRS_BSW_00337)*

The error classification is available in chapter 7 of the corresponding *BSW Module* specification.

[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)*

Example of *Development errors:*
For the first SW integration, the extended error detection and reporting are enabled for all BSW Modules. Detected errors like

- EEPROM address access out of valid range
- Sending on non-existent CAN channel
- API service called without former module initialization

are reported to the Development Error Tracer (Det). The calls to the API function of the Det are counted and logged for later evaluation. After successful software integration, the reporting is disabled.

Example of *Production errors:*

- NVRAM data block checksum error
- EEPROM cell write failure
- SPI device failure

7.2.3 Development Errors

7.2.3.1 Documentation

The SWS shall list the development errors in chapter 7 in accordance with the classification of SRS_BSW_00337 and SRS_BSW_00350.

[SWS_BSW_00201] Development error type
[Development error values are of Type *uint8.*]*()*

7.2.3.2 Configuration of Development Errors

[SWS_BSW_00202] Activation of Development Errors
[The activation of development errors is configurable (ON / OFF) at pre-compile time. The switch <MODULE PREFIX>_DEV_ERROR_DETECT (see chapter 10 of the respective module SWS) shall activate or deactivate the detection of all development errors.]*()*

[SWS_BSW_00203] API parameter checking
If the <MODULE PREFIX>_DEV_ERROR_DETECT 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_00338*).

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”).

The detection of *Development errors* is configurable at *Pre-compile time* for every single *BSW Module* (SRS_BSW_00338). The configuration parameter is specified in chapter 10 of the respective *BSW Module*. Its name is formed in the following way (SRS_BSW_00350):

```
<Mip>DevErrorDetect{<MIP>_DEV_ERROR_DETECT}
```

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.

7.2.3.3 Reporting Development errors

If 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 *Development error tracer* (*Det*) using the service `Det_ReportError`.]*(SRS_BSW_00338)*

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

See chapter 7.2.3 – “Development Errors” for more information about activation and deactivation of *Development error* detection. See the Specification of *Det[16]* for more information about the service `Det_ReportError`.

7.2.4 Production Errors and Extended Production Errors

7.2.4.1 Production errors and extended production errors

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

7.2.4.2 Configuration of Production Errors and Extended Production Errors

[SWS_BSW_00205] Detection of 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.]*()*

7.2.4.3 Reporting Production Errors and Extended Production Errors

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_ReportErrorStatus` is used:

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

Note that the configuration of Production errors is optional in the ECU Configuration of the BSW Modules.Note that not only error events from failed conditions checking may be reported. The
BSW Module specification shall also require reporting events when error-checking conditions are passed.

[SWS_BSW_00066] Report EventStatus to *Dem*
[For reporting an error state the following BSW specific interface of DEM shall be called:

```
void Dem_ReportErrorStatus(
     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 predebouncing `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)*

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. *[references to the requirements xxxx]*

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

[Note]
The callback service <Module>_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.

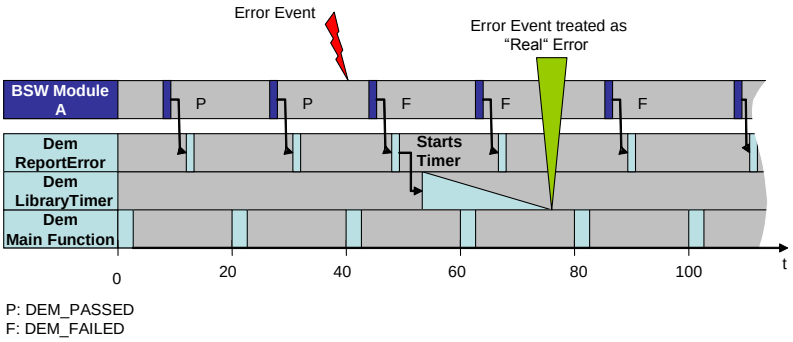
[SWS_BSW_00206] Only event-based error monitors shall implement the callback service
[<Module>_InitMonitorForEvent<EventName>.

[Note]
The BSW module Dem calls an implemented callback service <Module>_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 [<Module>_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 [<Module>_InitMonitorForEvent<EventName>, then the BSWMD shall specify a corresponding ServiceNeeds.]()

7.2.4.4 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*.

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.

8.2 Type definitions

[SWS_BSW_00146] Naming conventions for data types
⌈All data types defined by the *BSW Module* shall be labeled 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.	

⌋()

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.

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

[Data types used in Standard Interface and 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 Development 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	

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.

[SWS_BSW_00148] Naming convention for API services
⌈In the *BSW Module* implementation, all services from modules API shall be named in the following way:

```
<Mip>_<Sn>
```

Where *<Mip>* is the *Module implementation prefix* (SWS_BSW_00102) and <Sn> is the API *Service name*.⌋*(SRS_BSW_00310, SRS_BSW_00413, SRS_BSW_00347)*

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_TransmitFrame()`
- `Nm_RequestBusCommunication()`
- `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, especially the definitions which values of passed API parameters are invalid (e.g. check for maximum PduId value), are available in the according BSW Module specifications.

[SWS_BSW_00149] Do not pass function pointers as API parameter
⌈Function pointers shall not be passed as API parameter.⌋*(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 SWS_BSW_00148 and has `Init` as *Service name*.

Examples:

- `Can_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* signature has always the same pattern, where the main argument is the *Configuration pointer* for the selected configuration variant to be used. For instance:

```
void Eep_21_LDExt_Init(const Eep_ConfigType *ConfigPtr)
```

SWS_BSW_00047**[SWS_BSW_00185]** Return type of *initialization* functions
[The return type of *Initialization functions* is always `void`](SRS_BSW_00358).

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:

- If the initialization function does not need nor evaluate the passed argument, the *Configuration pointer* shall have a `NULL` value.
- If the initialization function requires the passed argument, the *Configuration pointer* shall be different from `NULL`.

If these conditions are not satisfied, a *Development error* with type "Invalid configuration set selection" shall be reported to *Development Error Tracer* (*Det*), see SWS_BSW_00151.](*SRS_BSW_00414, SRS_BSW_00400, SRS_BSW_00438*)

[SWS_BSW_00151] Name convention for error “Invalid configuration set selection”
[The name for the *Development error* “Invalid configuration set selection” should be formed in the following way:

```
<MIP>_E_INIT_FAILED
```

Where `<MIP>` is the Module implementation prefix of this BSW Module.](*SRS_BSW_00414*)

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

[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. If such a violation is detected a development error shall be raised.⌟()

Note: The name for the development errors for NULL pointer violations is usually <MIP>_E_PARAM_POINTER

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_00071] Set module initialization status
⌈The module initialization status must be set at the end of *Initialization function* execution. ⌋(*SRS_BSW_00450*)

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 SWS_BSW_00148 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 *UNINIT* after a call to its *De-Initialization function*.⌋(*SRS_BSW_00450*)

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_00168] *Get Version Information* function name
[The *Get Version Information* API name follows SWS_BSW_00148 and has `GetVersionInfo` as *Service name*.]*(SRS_BSW_00407)*

Example:
```
void Eep_GetVersionInfo (
     Std_VersionInfoType *versioninfo
)
```

[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 [13]):

- Vendor Id
- Module Id
- Vendor specific version number

[SWS_BSW_00051] Configuration parameter for enabling *Check Version Information* service
[The *BSW Module* shall provide a *Pre-compile time configuration parameter* for enabling or disabling the *Get Version Information API*. The configuration parameter name shall be formed in the following way:

```
<MIP>_VERSION_INFO_API
```

]*(SRS_BSW_00411)*

Example:
```
/* File: Eep_Cfg.h                                    */
#define EEP_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_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.

The function prototypes of the callback functions shall be provided in the *Callback header* file, see SWS_BSW_00026, chapter 5.1.7.3.

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.

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 Development 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_MainFunction(void)
```

b) Possible main processing functions of *FlexRay* driver:
```
void Fr_MainFunction_TxClst1(void)
void Fr_MainFunction_TxClst2(void)
void Fr_MainFunction_RxClst1(void)
void Fr_MainFunction_RxClst2(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, SRS_BSW_00376)*

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.

8.6.2 Optional Interfaces

The list of optional interfaces is specific for every *BSW Module*. Please refer to the corresponding *BSW Module* specification.

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.

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. E.g., Variant 1: only pre-compile time configuration parameters; Variant 2: mix of pre-compile- and post build time-configuration parameters. In one variant, a parameter can only be of one configuration class.

The possible configuration variants of a *BSW Module* are described in its specification. Each Variant has a unique name, which could be referenced to in later chapters. The maximum number of allowed variants is three. Note that each variant has its own requirement ID in the *BSW Module* specification.

10.1.3 Containers

Containers structure the 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 .*
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.*	
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).*
Scope			

[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

	• ***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 "- -".*

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.
L	*Loadable* - the configuration parameter shall be of configuration class *Post Build* and only one configuration parameter set resides in the ECU.
M	*Multiple* - the configuration parameter shall be of configuration class *Post Build* and is selected out of a set of multiple parameters by passing a dedicated pointer to the init function of the module.
--	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.

Two distinct names are specified:

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

- *Configuration parameter label specification*: It specifies the *Configuration parameter label* to be used for this parameter in implementation files, for instance: `EEP_NORMAL_WRITE_BLOCK_SIZE`.

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* and *Configuration parameter labels* shall start with the *Module implementation prefix* (SWS_BSW_00102) or its capitalized form. This is achieved by replacing the *Module abbreviation* term within the respective Configuration parameter name specification and Configuration parameter label specification through the *Module implementation* or its capitalized form.

Example:
The *Configuration parameter label specification* `EEP_NORMAL_WRITE_BLOCK_SIZE`, results in the derived *Configuration parameter label* `EEP_21_LDEXT_NORMAL_WRITE_BLOCK_SIZE` for the vendor with `VendorID==21` and with `vendorApiInfix==LDEXT`.

These rules allow configuration of multiple BSW driver modules from the same module type, even modules provided by same vendor.

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: Tp_Cfg.h                                */
/* Pre-compile time configuration                */

…
#define TP_USE_NORMAL_ADDRESSING          KTPOFF
#define TP_USE_NORMAL_FIXED_ADDRESSING    KTPOFF
#define TP_USE_EXTENDED_ADDRESSING        KTPON
…

/* File: Tp.c                                    */
…
#include "Tp_Cfg.h"
…
#if (TP_USE_NORMAL_ADDRESSING == KTPOFF)
…
#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. Usually those modules are BSW drivers.

[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* shall be pointed to by *Configuration pointers*. Only *EcuM* contains *Configuration pointers* to the *Post-build configuration data structure*.]*(SRS_BSW_00438)*

There are two types of post-build time configuration parameter sets: Loadable and selectable post-build time configurations.

Loadable post-build configuration sets are 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 loadable configuration, the memory layout of these parameters must be known:

[SWS_BSW_00159] Avoid pointer indirection to *Loadable post-build time configuration*
[If the *BSW Module* has *Loadable post-build time configuration* parameters, these parameters should be optimized in a way that pointer indirection is avoided, i.e. the configuration is always located in the same address.]*(SRS_BSW_00399)*

Selectable post-build time configurations enable the reuse of ECUs, as different configurations can be supported without reprogramming the ECU. These parameters will be selected from multiple sets of parameters after code has been loaded and started. This configuration is a data structure that contains the relevant parameter values.

[SWS_BSW_00058] Set selection of *Selectable post-build time configuration*
[If the *BSW Module* has *Selectable post-build time configuration* parameters, than one of several configurations shall be selected during module startup, i.e. within its *Initialization function*. See also SWS_BSW_00050.]*(SRS_BSW_00400)*

If there is at least one module with the configuration class “post build selectable” then the *EcuM* or the *BswM* (SWS_BSW_00150) will determine which pointer to the configuration parameters is required to be passed to the *Initialization functions* of *BSW Modules* (SWS_BSW_00050).

If there are no modules in the configuration class “post build selectable” but one or more modules are in the “post build” class then a fixed pointer will be passed to the *Initialization functions*.

If there is only one set of configuration data (i.e. there are no multiple configuration sets) the references can be resolved as constant pointers. The indirections shall be kept as simple as possible.

[SWS_BSW_00160] Reference pointer to Post-build time configurable data
[If the *BSW Module* operates on one Post-build time configurable data, then it shall use a reference (pointer) to an external configuration instance. This reference shall be read-only if only one configuration set is used.]*(SRS_BSW_00404, SRS_BSW_00405)*

Example:

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

/* File: ComM.h                                      */
…
/* Forward declaration:                              */
typedef struct ComM_ConfigType_Tag ComM_ConfigType;
extern void ComM(ComM_ConfigType * ComMConfigPtr);
```

...

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).

10.3 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***
<MIP>_VENDOR_ID	#define/uint16	Vendor ID (vendorId) of the dedicated implementation of this module according to the AUTOSAR vendor list. The ID is the same as in HIS Software Supplier Identifications [20].
<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: Can.h                                  */
…
/* Published information                        */
#define CAN_MODULE_ID_CFG          0x0050u
#define CAN_VENDOR_ID_CFG          0x002Bu
```

```
#define CAN_AR_RELEASE_MAJOR_VERSION_CFG   0x04u
#define CAN_AR_RELEASE_MINOR_VERSION_CFG   0x00u
#define CAN_AR_RELEASE_PATCH_VERSION_CFG   0x03u
#define CAN_SW_MAJOR_VERSION_CFG   0x01u
#define CAN_SW_MINOR_VERSION_CFG   0x02u
#define CAN_SW_PATCH_VERSION_CFG   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