Document Title	Specification of Compiler Abstraction
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	051
Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.3.1

Document Change History

Date	Release	Changed by	Change Description
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes • Clarification regarding module specific memory classes and global memory classes
2016-11-30	4.3.0	AUTOSAR Release Management	• Removed chapter 'Variants' • Removed obsolete elements
2015-07-31	4.2.2	AUTOSAR Release Management	• Cleanup the requirements traceability • Clarify the list of compiler symbols
2014-10-31	4.2.1	AUTOSAR Release Management	• The compiler symbol definitions are not allowed to contain any value behind the symbol • Rework the document structure in order to follow TMPS_SRS_SWS and replace hardcoded diagrams with artifacts • Remove all MISRA/ C/ C++ related statements and references • Correct the unresolved references that point in SRS_BSWGeneral
2013-10-31	4.1.2	AUTOSAR Release Management	• Editorial changes • Removed chapter(s) on change documentation

Document Change History

Date	Release	Changed by	Change Description
2013-03-15	4.1.1	AUTOSAR Administration	• Added abstraction macro CONSTP2FUNC for a constant pointer to a function • Improved consistency to Memory Mapping (several MemMap.h files) • Reworked Configuration Specification
2011-12-22	4.0.3	AUTOSAR Administration	• Added macros ‚FUNC_P2CONST‘ and ‘FUNC_P2VAR’ • Added pointer class ‘REGSPACE’ (for register access) • Updated the compiler symbols list
2010-09-30	3.1.5	AUTOSAR Administration	• Put more emphasize on SwComponentType’s name in SWS_COMPILER_00054, COMPILER044 • Corrected compiler used in the example (chapter 7.1.5) • Corrected include structure in the example (chapter 7.1.5)
2010-02-02	3.1.4	AUTOSAR Administration	• Compiler Abstraction has been extended to be suitable for Software Components • "STATIC" declaration keyword has been removed • The declaration keyword "LOCAL_INLINE" has been added for implementation of "static inline"-functions • Legal disclaimer revised
2008-08-13	3.1.1	AUTOSAR Administration	• Legal disclaimer revised

Document Change History			
Date	**Release**	**Changed by**	**Change Description**
2007-12-21	3.0.1	AUTOSAR Administration	• Keyword "_STATIC_" has been renamed to "STATIC" • Keyword "_INLINE_" has been renamed to "INLINE" • Keyword "TYPEDEF" has been added as empty memory qualifier for use in type definitions • Document meta information extended • Small layout adaptations made
2007-01-24	2.1.15	AUTOSAR Administration	• Add: COMPILER058 • Add: COMPILER057 • Change: SWS_COMPILER_00040 • Legal disclaimer revised • Release Notes added • “Advice for users” revised • “Revision Information” added
2006-05-16	2.0	AUTOSAR Administration	• Initial Release

Disclaimer

Table of Contents

1 Introduction and functional overview

This document specifies macros for the abstraction of compiler specific keywords used for addressing data and code within declarations and definitions.

Mainly compilers for 16-bit platforms (e.g. Cosmic and Metrowerks for S12X or Tasking for ST10) are using special keywords to cope with properties of the microcontroller architecture caused by the limited 16 bit addressing range. Features like paging and extended addressing (to reach memory beyond the 64k border) are not chosen automatically by the compiler, if the memory model is not adjusted to ‘large’ or ‘huge’. The location of data and code has to be selected explicitly by special keywords. Those keywords, if directly used within the source code, would make it necessary to port the software to each new microcontroller family and would prohibit the requirement of platform independency of source code.

If the memory model is switched to ‘large’ or ‘huge’ by default (to circumvent these problems) the project will suffer from an increased code size.

This document specifies a three-step concept:

1. The file Compiler.h provides macros for the encapsulation of definitions and declarations.

2. Each single module has to distinguish between at least the following different memory classes and pointer classes. Each of these classes is represented by a define.
3. The file Compiler_Cfg.h allows to configure these defines with the appropriate compiler specific keywords according to the modules description and memory set-up of the build scenario.

2 Acronyms and abbreviations

Acronyms and abbreviations that have a local scope are not contained in the AUTOSAR glossary. These must appear in a local glossary.

Acronym:	***Description:***
Large, huge	Memory model configuration of the microcontroller's compiler. By default, all access mechanisms are using extended/paged addressing. Some compilers are using the term 'huge' instead of 'far'.
Tiny, small	Memory model configuration of the microcontroller's compiler. By default, all access mechanisms are using normal addressing. Only data and code within the addressing range of the platform's architecture is reachable (e.g. 64k on a 16 bit architecture).
far	Compiler keyword for extended/paged addressing scheme (for data and code that may be outside the normal addressing scheme of the platform's architecture).
near	Compiler keyword for normal addressing scheme (for data and code that is within the addressing range of the platform's architecture).

3 Related documentation

3.1 Input documents

[1] List of Basic Software Modules,
AUTOSAR_TR_BSWModuleList.pdf

[2] General Requirements on Basic Software Modules,
AUTOSAR_SRS_BSWGeneral.pdf

[3] Layered Software Architecture,
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[4] Specification of ECU Configuration,
AUTOSAR_TPS_ECUConfiguration.pdf

[5] Cosmic C Cross Compiler User's Guide for Motorola MC68HC12,V4.5

[6] ARM ADS compiler manual

[7] GreenHills MULTI for V850 V4.0.5:
Building Applications for Embedded V800, V4.0, 30.1.2004

[8] TASKING for ST10 V8.5:
C166/ST10 v8.5 C Cross-Compiler User's Manual, V5.16
C166/ST10 v8.5 C Cross-Assembler, Linker/Locator, Utilities User's Manual, V5.16

[9] Wind River (Diab Data) for PowerPC Version 5.2.1:
Wind River Compiler for Power PC - Getting Started, Edition 2, 8.5.2004
Wind River Compiler for Power PC - User's Guide, Edition 2, 11.5.2004

[10] TASKING for TriCore TC1796 V2.0R1:
TriCore v2.0 C Cross-Compiler, Assembler, Linker User's Guide, V1.2

[11] Metrowerks CodeWarrior 4.0 for Freescale HC9S12X/XGATE (V5.0.25):
Motorola HC12 Assembler, 2.6.2004
Motorola HC12 Compiler, 2.6.2004
Smart Linker, 2.4.2004

[12] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral.pdf

[13] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

3.2 Related specification

AUTOSAR provides a General Specification on Basic Software modules [12] (SWS BSW General), which is also valid for Compiler Abstraction.

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

4 Constraints and assumptions

4.1 Limitations

During specification of abstraction and validation of concept, the compilers listed in chapter 3.1 has been considered. If any other compiler requires keywords that cannot be mapped to the mechanisms described in this specification this compiler will not be supported by AUTOSAR. In this case, the compiler vendor has to adapt its compiler.

If the physically existing memory is larger than the logically addressable memory in either code space or data space and more than the logically addressable space is used, logical addresses have to be re-used. The C language (and other languages as well) cannot cope with this situation.

4.2 Applicability to car domains

No restrictions.

4.3 Applicability to safety related environments

No restrictions. The compiler abstraction file does not implement any functionality, only symbols and macros.

5 Dependencies to other modules

[SWS_COMPILER_00048] ⌈ The SWS Compiler Abstraction is applicable for each AUTOSAR basic software module and application software components. Therefore, the implementation of the memory class (memclass) and pointer class (ptrclass) macro parameters (see SWS_COMPILER_00040) shall fulfill the implementation and configuration specific needs of each software module in a specific build scenario. ⌋ (SRS_BSW_00328, SRS_BSW_00384)

5.1 File structure

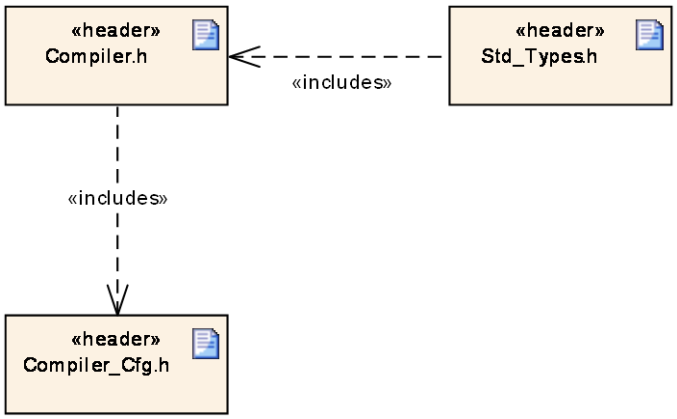

Figure 1: Include structure of Compiler.h

The following notes shall describe the connections to modules, which are indirectly linked to each other.

Note 1: The compiler abstraction is used to configure the reachability of elements (pointers, variables, function etc.).

Note 2: The memory mapping is used to perform the sectioning of memory. The user can define sections for optimizing the source code.

Note 3: The linker settings are responsible with the classification which elements are assigned to which memory section.

6 Requirements traceability

The requirements above are derived from SRS_BSWGeneral.

Requirement	Description	Satisfied by
SRS_BSW_00003	All software modules shall provide version and identification information	SWS_COMPILER_00001
SRS_BSW_00004	All Basic SW Modules shall perform a pre-processor check of the versions of all imported include files	SWS_COMPILER_00999
SRS_BSW_00005	Modules of the µC Abstraction Layer (MCAL) may not have hard coded horizontal interfaces	SWS_COMPILER_00999
SRS_BSW_00006	The source code of software modules above the µC Abstraction Layer (MCAL) shall not be processor and compiler dependent.	SWS_COMPILER_00010, SWS_COMPILER_00035, SWS_COMPILER_00036
SRS_BSW_00007	All Basic SW Modules written in C language shall conform to the MISRA C 2012 Standard.	SWS_COMPILER_00999
SRS_BSW_00009	All Basic SW Modules shall be documented according to a common standard.	SWS_COMPILER_00999
SRS_BSW_00010	The memory consumption of all Basic SW Modules shall be documented for a defined configuration for all supported platforms.	SWS_COMPILER_00999
SRS_BSW_00158	All modules of the AUTOSAR Basic Software shall strictly separate configuration from implementation	SWS_COMPILER_00999
SRS_BSW_00161	The AUTOSAR Basic Software shall provide a microcontroller abstraction layer which provides a standardized interface to higher software layers	SWS_COMPILER_00999
SRS_BSW_00162	The AUTOSAR Basic Software shall provide a hardware abstraction layer	SWS_COMPILER_00999
SRS_BSW_00164	The Implementation of interrupt service routines shall be done by the Operating System, complex drivers or modules	SWS_COMPILER_00999
SRS_BSW_00167	All AUTOSAR Basic Software	SWS_COMPILER_00999

	Modules shall provide configuration rules and constraints to enable plausibility checks	
SRS_BSW_00168	SW components shall be tested by a function defined in a common API in the Basis-SW	SWS_COMPILER_00999
SRS_BSW_00170	The AUTOSAR SW Components shall provide information about their dependency from faults, signal qualities, driver demands	SWS_COMPILER_00999
SRS_BSW_00171	Optional functionality of a Basic-SW component that is not required in the ECU shall be configurable at pre-compile-time	SWS_COMPILER_00999
SRS_BSW_00172	The scheduling strategy that is built inside the Basic Software Modules shall be compatible with the strategy used in the system	SWS_COMPILER_00999
SRS_BSW_00300	All AUTOSAR Basic Software Modules shall be identified by an unambiguous name	SWS_COMPILER_00999
SRS_BSW_00301	All AUTOSAR Basic Software Modules shall only import the necessary information	SWS_COMPILER_00999
SRS_BSW_00302	All AUTOSAR Basic Software Modules shall only export information needed by other modules	SWS_COMPILER_00999
SRS_BSW_00305	Data types naming convention	SWS_COMPILER_00999
SRS_BSW_00306	AUTOSAR Basic Software Modules shall be compiler and platform independent	SWS_COMPILER_00010, SWS_COMPILER_00035, SWS_COMPILER_00036, SWS_COMPILER_00058
SRS_BSW_00307	Global variables naming convention	SWS_COMPILER_00999
SRS_BSW_00308	AUTOSAR Basic Software Modules shall not define global data in their header files, but in the C file	SWS_COMPILER_00999
SRS_BSW_00309	All AUTOSAR Basic Software Modules shall indicate all global data with read-only purposes by explicitly assigning the const keyword	SWS_COMPILER_00023, SWS_COMPILER_00999
SRS_BSW_00310	API naming convention	SWS_COMPILER_00999
SRS_BSW_00312	Shared code shall be reentrant	SWS_COMPILER_00999
SRS_BSW_00314	All internal driver modules shall	SWS_COMPILER_00999

	separate the interrupt frame definition from the service routine	
SRS_BSW_00323	All AUTOSAR Basic Software Modules shall check passed API parameters for validity	SWS_COMPILER_00999
SRS_BSW_00325	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	SWS_COMPILER_00999
SRS_BSW_00327	Error values naming convention	SWS_COMPILER_00999
SRS_BSW_00328	All AUTOSAR Basic Software Modules shall avoid the duplication of code	SWS_COMPILER_00048
SRS_BSW_00330	It shall be allowed to use macros instead of functions where source code is used and runtime is critical	SWS_COMPILER_00999
SRS_BSW_00331	All Basic Software Modules shall strictly separate error and status information	SWS_COMPILER_00999
SRS_BSW_00333	For each callback function it shall be specified if it is called from interrupt context or not	SWS_COMPILER_00999
SRS_BSW_00334	All Basic Software Modules shall provide an XML file that contains the meta data	SWS_COMPILER_00999
SRS_BSW_00335	Status values naming convention	SWS_COMPILER_00999
SRS_BSW_00336	Basic SW module shall be able to shutdown	SWS_COMPILER_00999
SRS_BSW_00339	Reporting of production relevant error status	SWS_COMPILER_00999
SRS_BSW_00341	Module documentation shall contains all needed informations	SWS_COMPILER_00999
SRS_BSW_00342	It shall be possible to create an AUTOSAR ECU out of modules provided as source code and modules provided as object code, even mixed	SWS_COMPILER_00999
SRS_BSW_00343	The unit of time for specification and configuration of Basic SW modules shall be preferably in physical time unit	SWS_COMPILER_00999
SRS_BSW_00344	BSW Modules shall support link-time configuration	SWS_COMPILER_00999
SRS_BSW_00346	All AUTOSAR Basic Software Modules shall provide at least a basic set of module files	SWS_COMPILER_00999

SRS_BSW_00347	A Naming seperation of different instances of BSW drivers shall be in place	SWS_COMPILER_00050
SRS_BSW_00348	All AUTOSAR standard types and constants shall be placed and organized in a standard type header file	SWS_COMPILER_00003, SWS_COMPILER_00004
SRS_BSW_00350	All AUTOSAR Basic Software Modules shall allow the enabling/disabling of detection and reporting of development errors.	SWS_COMPILER_00999
SRS_BSW_00351	Encapsulation of compiler specific methods to map objects	SWS_COMPILER_00999
SRS_BSW_00353	All integer type definitions of target and compiler specific scope shall be placed and organized in a single type header	SWS_COMPILER_00999
SRS_BSW_00357	For success/failure of an API call a standard return type shall be defined	SWS_COMPILER_00999
SRS_BSW_00358	The return type of init() functions implemented by AUTOSAR Basic Software Modules shall be void	SWS_COMPILER_00999
SRS_BSW_00359	All AUTOSAR Basic Software Modules callback functions shall avoid return types other than void if possible	SWS_COMPILER_00999
SRS_BSW_00360	AUTOSAR Basic Software Modules callback functions are allowed to have parameters	SWS_COMPILER_00999
SRS_BSW_00361	All mappings of not standardized keywords of compiler specific scope shall be placed and organized in a compiler specific type and keyword header	13, SWS_COMPILER_00003, SWS_COMPILER_00004, SWS_COMPILER_00006, SWS_COMPILER_00013, SWS_COMPILER_00026, SWS_COMPILER_00031, SWS_COMPILER_00032, SWS_COMPILER_00039, SWS_COMPILER_00040, SWS_COMPILER_00041, SWS_COMPILER_00046, SWS_COMPILER_00047, SWS_COMPILER_00053, SWS_COMPILER_00055, SWS_COMPILER_00057, SWS_COMPILER_00060, SWS_COMPILER_00061, SWS_COMPILER_00062, SWS_COMPILER_00063, SWS_COMPILER_00064, SWS_COMPILER_00065

SRS_BSW_00369	All AUTOSAR Basic Software Modules shall not return specific development error codes via the API	SWS_COMPILER_00999
SRS_BSW_00371	The passing of function pointers as API parameter is forbidden for all AUTOSAR Basic Software Modules	SWS_COMPILER_00999
SRS_BSW_00373	The main processing function of each AUTOSAR Basic Software Module shall be named according the defined convention	SWS_COMPILER_00999
SRS_BSW_00374	All Basic Software Modules shall provide a readable module vendor identification	SWS_COMPILER_00030
SRS_BSW_00375	Basic Software Modules shall report wake-up reasons	SWS_COMPILER_00999
SRS_BSW_00377	A Basic Software Module can return a module specific types	SWS_COMPILER_00999
SRS_BSW_00378	AUTOSAR shall provide a boolean type	SWS_COMPILER_00999
SRS_BSW_00380	Configuration parameters being stored in memory shall be placed into separate c-files	SWS_COMPILER_00999
SRS_BSW_00383	The Basic Software Module specifications shall specify which other configuration files from other modules they use at least in the description	SWS_COMPILER_00999
SRS_BSW_00384	The Basic Software Module specifications shall specify at least in the description which other modules they require	SWS_COMPILER_00048
SRS_BSW_00385	List possible error notifications	SWS_COMPILER_00999
SRS_BSW_00386	The BSW shall specify the configuration for detecting an error	SWS_COMPILER_00999
SRS_BSW_00388	Containers shall be used to group configuration parameters that are defined for the same object	SWS_COMPILER_00040, SWS_COMPILER_00999
SRS_BSW_00389	Containers shall have names	SWS_COMPILER_00999
SRS_BSW_00390	Parameter content shall be unique within the module	SWS_COMPILER_00999
SRS_BSW_00392	Parameters shall have a type	SWS_COMPILER_00999
SRS_BSW_00393	Parameters shall have a range	SWS_COMPILER_00999
SRS_BSW_00394	The Basic Software Module specifications shall specify the scope of the configuration parameters	SWS_COMPILER_00999

SRS_BSW_00395	The Basic Software Module specifications shall list all configuration parameter dependencies	SWS_COMPILER_00999
SRS_BSW_00396	The Basic Software Module specifications shall specify the supported configuration classes for changing values and multiplicities for each parameter/container	SWS_COMPILER_00999
SRS_BSW_00398	The link-time configuration is achieved on object code basis in the stage after compiling and before linking	SWS_COMPILER_00999
SRS_BSW_00399	Parameter-sets shall be located in a separate segment and shall be loaded after the code	SWS_COMPILER_00999
SRS_BSW_00400	Parameter shall be selected from multiple sets of parameters after code has been loaded and started	SWS_COMPILER_00999
SRS_BSW_00401	Documentation of multiple instances of configuration parameters shall be available	SWS_COMPILER_00999
SRS_BSW_00403	The Basic Software Module specifications shall specify for each parameter/container whether it supports different values or multiplicity in different configuration sets	SWS_COMPILER_00999
SRS_BSW_00404	BSW Modules shall support post-build configuration	SWS_COMPILER_00059, SWS_COMPILER_00999
SRS_BSW_00405	BSW Modules shall support multiple configuration sets	SWS_COMPILER_00999
SRS_BSW_00406	A static status variable denoting if a BSW module is initialized shall be initialized with value 0 before any APIs of the BSW module is called	SWS_COMPILER_00999
SRS_BSW_00407	Each BSW module shall provide a function to read out the version information of a dedicated module implementation	SWS_COMPILER_00999
SRS_BSW_00408	All AUTOSAR Basic Software Modules configuration parameters shall be named according to a specific naming rule	SWS_COMPILER_00999
SRS_BSW_00409	All production code error ID symbols are defined by the Dem module and shall be	SWS_COMPILER_00999

	retrieved by the other BSW modules from Dem configuration	
SRS_BSW_00410	Compiler switches shall have defined values	SWS_COMPILER_00999
SRS_BSW_00411	All AUTOSAR Basic Software Modules shall apply a naming rule for enabling/disabling the existence of the API	SWS_COMPILER_00999
SRS_BSW_00412	References to c-configuration parameters shall be placed into a separate h-file	SWS_COMPILER_00999
SRS_BSW_00413	An index-based accessing of the instances of BSW modules shall be done	SWS_COMPILER_00999
SRS_BSW_00414	Init functions shall have a pointer to a configuration structure as single parameter	SWS_COMPILER_00999
SRS_BSW_00415	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	SWS_COMPILER_00999
SRS_BSW_00416	The sequence of modules to be initialized shall be configurable	SWS_COMPILER_00999
SRS_BSW_00417	Software which is not part of the SW-C shall report error events only after the DEM is fully operational.	SWS_COMPILER_00999
SRS_BSW_00419	If a pre-compile time configuration parameter is implemented as "const" it should be placed into a separate c-file	SWS_COMPILER_00999
SRS_BSW_00422	Pre-de-bouncing of error status information is done within the DEM	SWS_COMPILER_00999
SRS_BSW_00423	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	SWS_COMPILER_00999
SRS_BSW_00424	BSW module main processing functions shall not be allowed to enter a wait state	SWS_COMPILER_00999
SRS_BSW_00425	The BSW module description template shall provide means to model the defined trigger conditions of schedulable objects	SWS_COMPILER_00999
SRS_BSW_00426	BSW Modules shall ensure data consistency of data which is shared between BSW	SWS_COMPILER_00999

	modules	
SRS_BSW_00427	ISR functions shall be defined and documented in the BSW module description template	SWS_COMPILER_00999
SRS_BSW_00428	A BSW module shall state if its main processing function(s) has to be executed in a specific order or sequence	SWS_COMPILER_00999
SRS_BSW_00429	Access to OS is restricted	SWS_COMPILER_00999
SRS_BSW_00432	Modules should have separate main processing functions for read/receive and write/transmit data path	SWS_COMPILER_00999
SRS_BSW_00433	Main processing functions are only allowed to be called from task bodies provided by the BSW Scheduler	SWS_COMPILER_00999
SRS_BSW_00448	Module SWS shall not contain requirements from Other Modules	SWS_COMPILER_00999
SRS_BSW_00449	BSW Service APIs used by Autosar Application Software shall return a Std_ReturnType	SWS_COMPILER_00999
SRS_BSW_00452	Classification of runtime errors	SWS_COMPILER_00999
SRS_BSW_00453	BSW Modules shall be harmonized	SWS_COMPILER_00999
SRS_BSW_00454	An alternative interface without a parameter of category DATA_REFERENCE shall be available.	SWS_COMPILER_00999
SRS_BSW_00456	- A Header file shall be defined in order to harmonize BSW Modules	SWS_COMPILER_00054, SWS_COMPILER_00999
SRS_BSW_00457	- Callback functions of Application software components shall be invoked by the Basis SW	SWS_COMPILER_00999
SRS_BSW_00458	Classification of production errors	SWS_COMPILER_00999
SRS_BSW_00459	It shall be possible to concurrently execute a service offered by a BSW module in different partitions	SWS_COMPILER_00999
SRS_BSW_00461	Modules called by generic modules shall satisfy all interfaces requested by the generic module	SWS_COMPILER_00999
SRS_BSW_00462	All Standardized Autosar Interfaces shall have unique requirement Id / number	SWS_COMPILER_00999
SRS_BSW_00464	File names shall be considered	SWS_COMPILER_00004,

	case sensitive regardless of the filesystem in which they are used	SWS_COMPILER_00055
SRS_BSW_00466	Classification of extended production errors	SWS_COMPILER_00999
SRS_BSW_00469	Fault detection and healing of production errors and extended production errors	SWS_COMPILER_00999
SRS_BSW_00470	Execution frequency of production error detection	SWS_COMPILER_00999
SRS_BSW_00471	Do not cause dead-locks on detection of production errors - the ability to heal from previously detected production errors	SWS_COMPILER_00999
SRS_BSW_00472	Avoid detection of two production errors with the same root cause.	SWS_COMPILER_00999
SRS_BSW_00473	Classification of transient faults	SWS_COMPILER_00999
SRS_BSW_00478	Timing limits of main functions	SWS_COMPILER_00999
SRS_BSW_00479	Interfaces for handling request from external devices	SWS_COMPILER_00999
SRS_BSW_00480	NullPointer Errors shall follow a naming rule	SWS_COMPILER_00051, SWS_COMPILER_00999
SRS_BSW_00481	Invalid configuration set selection errors shall follow a naming rule	SWS_COMPILER_00999
SRS_BSW_00482	Get Version Informationfunction shall follow a naming rule	SWS_COMPILER_00999
SRS_BSW_00483	BSW Modules shall handle buffer alignments internally	SWS_COMPILER_00999

7 Functional specification

7.1 General behavior

[SWS_COMPILER_00003] ⌈ For each compiler and platform an own compiler abstraction has to be provided. ⌋ (SRS_BSW_00348, SRS_BSW_00361)

7.1.1 List of Compiler symbols

The following table defines target compiler symbols according to SWS_COMPILER_00010. This table contains only examples and is not listing all the possible compilers supported by AUTOSAR!

Platform	***Compiler***	***Compiler symbol***
S12X	Code Warrior	_CODEWARRIOR_C_S12X_
S12X	Cosmic	_COSMIC_C_S12X_
TC1796/ TC1766	Tasking	_TASKING_C_TRICORE_
ST10	Tasking	_TASKING_C_ST10_
ST30	ARM Developer Suite	_ADS_C_ST30_
V850	Greenhills	_GREENHILLS_C_V850_
MPC5554	Diab Data	_DIABDATA_C_ESYS_
TMS470	Texas Instruments	_TEXAS_INSTRUMENTS_C_TMS470_
ARM	Texas Instruments	_TEXAS_INSTRUMENTS_C_ARM_

Note: In order to avoid incompatibilities and/ or inconsistencies, the compiler symbol definitions are not allowed to contain any value behind the symbol.

7.1.2 Requirements on implementations using compiler abstraction

[SWS_COMPILER_00040] ⌈ Each AUTOSAR software module and application software component shall support the distinction of at least the following different memory classes and pointer classes. ⌋ (SRS_BSW_00361, SRS_BSW_00388)

It is allowed to add module specific memory classes and pointer classes as they are mapped and thus are configurable within the Compiler_Cfg.h file.

<**PREFIX**> is

- composed according <snp>[_<vi>_<ai>] for basic software modules where
  - <snp> is the *Section Name Prefix* which shall be the `BswModuleDescription`'s `shortName` converted in upper case letters if no SectionNamePrefix is defined for the MemorySection in the Basic Software Module Description or Software Component Description.
  - <snp> shall be the symbol of the Section NamePrefix associated to the MemorySection if a SectionNamePrefix is defined for the MemorySection.
  - <vi> is the `vendorId` of the BSW module
  - <ai> is the `vendorApiInfix` of the BSW module

  The sub part in squared brackets [_<vi>_<ai>] is omitted if no `vendorApiInfix` is defined for the Basic Software Module which indicates that it does not use multiple instantiation.
- the `shortName` of the software component type for software components (case sensitive)

<INIT_POLICY> is the initialization policy of variables. Possible values are:

- NO_INIT: Used for variables that are never cleared and never initialized.
- CLEARED: Used for variables that are cleared to zero after every reset.
- POWER_ON_CLEARED: Used for variables that are cleared to zero only after power on reset.
- INIT: Used for variables that are initialized with values after every reset.
- POWER_ON_INIT: Used for variables that are initialized with values only after power on reset.

Memory type	***Syntax of memory class (memclass) and pointer class (ptrclass) macro parameter***	***Comments***	***Located in***
Code	`<PREFIX>_CODE[_<PERIOD>]`	To be used for code. PERIOD is the typical period time value and unit of the ExecutableEntitys in this MemorySection. The name part [_<PERIOD>] is optional. units are: US microseconds MS milli second S second For example: 100US, 400US, 1MS, 5MS, 10MS, 20MS, 100MS, 1S Please note that deviations from this typical period time are possible due to integration decisions (e.g. RTEEvent To Task Mapping). Further, in special modes of the ECU the code may be scheduled with a higher or lower period.	Compiler_Cfg.h

Memory type	*Syntax of memory class (memclass) and pointer class (ptrclass) macro parameter*	*Comments*	*Located in*
Code	<PREFIX>_<CN>_CODE	To be used for callout code. <CN> is the callback name (including module reference) written in uppercase letters.	
Code	<PREFIX>_CODE_FAST	To be used for code that shall go into fast code memory segments. The FAST sections should be used when the execution does not happen in a well-defined period time but with the knowledge of high frequent access and /or high execution time, for example, a callback for a frequent notification.	
Code	<PREFIX>_CODE_SLOW	To be used for code that shall go into slow code memory segments. The SLOW sections should be used when the execution does not happen in a well-defined period time but with the knowledge of low frequent access, for example, a callback in case of seldom error.	
Constants	<PREFIX>_CONST	To be used for global or static constants.	
Constants	<PREFIX>_CALIB	To be used for calibration constants.	
Constants	<PREFIX>_CONFIG_DATA	To be used for module configuration constants.	
Constants	<PREFIX>_CONST_SAVED_RECOVERY_ZONE<X>	To be used for ROM buffers of variables saved in non-volatile memory. X shall be replaced with the number of bytes (i.e. 8,16 or 32).	
Pointer	<PREFIX>_APPL_DATA	To be used for references on application data (expected to be in RAM or ROM) passed via API	
Pointer	<PREFIX>_APPL_CONST	To be used for references on application constants (expected to be certainly in ROM, for instance pointer of Init-function) passed via API	
Pointer	REGSPACE	To be used for pointers to registers (e.g. `static volatile CONSTP2VAR(uint16, PWM_CONST, REGSPACE)`).	
Variables	<PREFIX>_VAR_<INIT_POLICY>	To be used for all global or static variables.	

Memory type	*Syntax of memory class (memclass) and pointer class (ptrclass) macro parameter*	*Comments*	*Located in*
Variables	<PREFIX>_VAR_FAST_<INIT_POLICY>	To be used for all global or static variables that have at least one of the following properties: • accessed bitwise • frequently used • high number of accesses in source code Some platforms allow the use of bit instructions for variables located in this specific RAM area as well as shorter addressing instructions. This saves code and runtime.	
Variables	<PREFIX>_VAR_SLOW_<INIT_POLICY>	To be used for all infrequently accessed global or static variables.	
Variables	<PREFIX>_INTERNAL_VAR_<INIT_POLICY>	To be used for global or static variables which are accessible from a calibration tool.	
Variables	<PREFIX>_VAR_SAVED_ZONE<X>	To be used for RAM buffers of variables saved in non-volatile memory. X shall be replaced with the number of bytes (i.e. 8,16 or 32).	
Variables	AUTOMATIC	To be used for local non static variables	Compiler.h
Type Definitions	TYPEDEF	To be used in type definitions, where no memory qualifier can be specified.	Compiler.h

For the memory classes that have the form <PREFIX>_<NAME>, one can specify the part <NAME> in the the MemorySections of a Basic Software Module Description or Software Component Description as follows. This is especially required for generated code:

- <NAME> is the shortName (case sensitive) of the SwAddrMethod referred from the MemorySection if if the MemorySection has no memClassSymbol attribute defined.
- Only for Basic Software: <NAME> is the memClassSymbol (case sensitive) of the MemorySection if this attribute is defined.

⌋ ()

[SWS_COMPILER_00041] ⌈ Each AUTOSAR software module and application software component shall wrap declaration and definition of code, variables, constants and pointer types using the following keyword macros: ⌋ (SRS_BSW_00361)

For instance:
native C-API:

```
Std_ReturnType Spi_SetupBuffers
(
    Spi_ChannelType       Channel,
    const Spi_DataType   *SrcDataBufferPtr,
    Spi_DataType         *DesDataBufferPtr,
    Spi_NumberOfDataType  Length
);
```

is encapsulated:

```
FUNC(Std_ReturnType, SPI_CODE) Spi_SetupBuffers
(
    Spi_ChannelType        Channel,
    P2CONST(Spi_DataType, AUTOMATIC, SPI_APPL_DATA) SrcDataBufferPtr,
    P2VAR(Spi_DataType, AUTOMATIC, SPI_APPL_DATA,)  DesDataBufferPtr,
    Spi_NumberOfDataType   Length
);
```

7.1.3 Contents of Compiler.h

[SWS_COMPILER_00004] ⌈ The file name of the compiler abstraction shall be 'Compiler.h'. ⌋ (SRS_BSW_00348, SRS_BSW_00361, SRS_BSW_00464)

[SWS_COMPILER_00053] ⌈ The file Compiler.h shall contain the definitions and macros specified in chapter 7.1.5. Those are fix for one specific compiler and platform. ⌋ (SRS_BSW_00361)

[SWS_COMPILER_00005] ⌈ If a compiler does not require or support the usage of special keywords; the corresponding macros specified by this specification shall be provided as empty definitions or definitions without effect.
Example:

```
#define FUNC(type, memclass) type
/* not required for DIABDATA */
```

⌋

[SWS_COMPILER_00010] ⌈ The compiler abstraction shall define a symbol for the target compiler according to the following naming convention:
_<COMPILERNAME>_C_<PLATFORMNAME>_

Note 1: In order to avoid incompatibilities and/ or inconsistencies, the compiler symbol definitions are not allowed to contain any value behind the symbol.

Note 2: These defines can be used to switch between different implementations for different compilers, e.g.

- inline assembler fragments in drivers
- special pragmas for memory alignment control
- localization of function calls
- adaptions to memory models ⌋ (SRS_BSW_00306, SRS_BSW_00006)

List of symbols: see chapter 7.1.1.

[SWS_COMPILER_00030] ⌈ "Compiler.h" shall provide information of the supported compiler vendor and the applicable compiler version. ⌋ (SRS_BSW_00374)

[SWS_COMPILER_00035] ⌈ The macro parameters memclass and ptrclass shall not be filled with the compiler specific keywords but with one of the configured values in SWS_COMPILER_00040. ⌋ (SRS_BSW_00306, SRS_BSW_00006)

The rationale is that the module's implementation shall not be affected when changing a variable's, a pointer's or a function's storage class.

[SWS_COMPILER_00036] ⌈ C forbids the use of the far/near-keywords on function local variables (auto-variables). For this reason when using the macros below to allocate a pointer on stack, the memclass-parameter shall be set to AUTOMATIC. ⌋ (SRS_BSW_00306, SRS_BSW_00006)

[SWS_COMPILER_00047] ⌈ The Compiler.h header file shall protect itself against multiple inclusions.
For instance:

```
#ifndef COMPILER_H
   #define COMPILER_H
    /* implementation of Compiler.h */
    ...
   #endif /* COMPILER_H */
```

There may be only comments outside of the ifndef - endif bracket. ⌋ (SRS_BSW_00361)

[SWS_COMPILER_00050] ⌈ It is allowed to extend the Compiler Abstraction header with vendor specific extensions. Vendor specific extended elements shall contain the AUTOSAR Vendor ID in the name. ⌋ (SRS_BSW_00347)

7.1.4 Contents of Compiler_Cfg.h

[SWS_COMPILER_00055] ⌈ The file Compiler_Cfg.h shall contain the module/component specific parameters (`ptrclass` and `memclass`) that are passed to the macros defined in Compiler.h. See SWS_COMPILER_00040 for memory types and required syntax. ⌋ (SRS_BSW_00361, SRS_BSW_00464)

[SWS_COMPILER_00054] ⌈ Module specific extended elements shall contain the module abbreviation of the BSW module in the name. Application software component specific extended elements shall contain the Software Component Type's name. ⌋ (SRS_BSW_00456)

7.1.5 Comprehensive example

This example shows for a single API function where which macro is defined, used and configured.
Module: Eep
API function: Eep_Read
Platform: S12X
Compiler: Cosmic

File Eep.c:

```
#include "Std_Types.h"  /* This includes also Compiler.h */

FUNC(Std_ReturnType, EEP_CODE) Eep_Read
              (
                  Eep_AddressType EepromAddress,
                  P2VAR(uint8, AUTOMATIC, EEP_APPL_DATA) DataBufferPtr,
                  Eep_LengthType Length
              )
```

File Compiler.h:

```
#include "Compiler_Cfg.h"

#define AUTOMATIC
#define FUNC(rettype, memclass) rettype memclass
#define P2VAR(ptrtype, memclass, ptrclass)  ptrclass ptrtype * memclass
```

File Compiler_Cfg.h:

```
#define EEP_CODE
#define EEP_APPL_DATA @far  /* RAM blocks of NvM are in banked RAM */
```

What are the dependencies?
`EEP_APPL_DATA` is defined as ‘far’. This means that the pointers to the RAM blocks managed by the NVRAM Manager have to be defined as ‘far’ also. The application can locate RAM mirrors in banked RAM but also in non-banked RAM. The mapping of the RAM blocks to banked RAM is done in <Mip>_MemMap.h (see [12] for more information on <Mip>).

Because the pointers are also passed via Memory Interface and EEPROM Abstraction, their pointer and memory classes must also fit to `EEP_APPL_DATA`.

What would be different on a 32-bit platform?
Despite the fact that only the S12X has an internal EEPROM, the only thing that would change in terms of compiler abstraction are the definitions in Compiler_Cfg.h. They would change to empty defines:

```
#define EEP_CODE
#define EEP_APPL_DATA
```

7.1.6 Proposed process

To allow development and integration within a multi supplier environment a certain delivery process is indispensable. The following description can be seen as proposal:

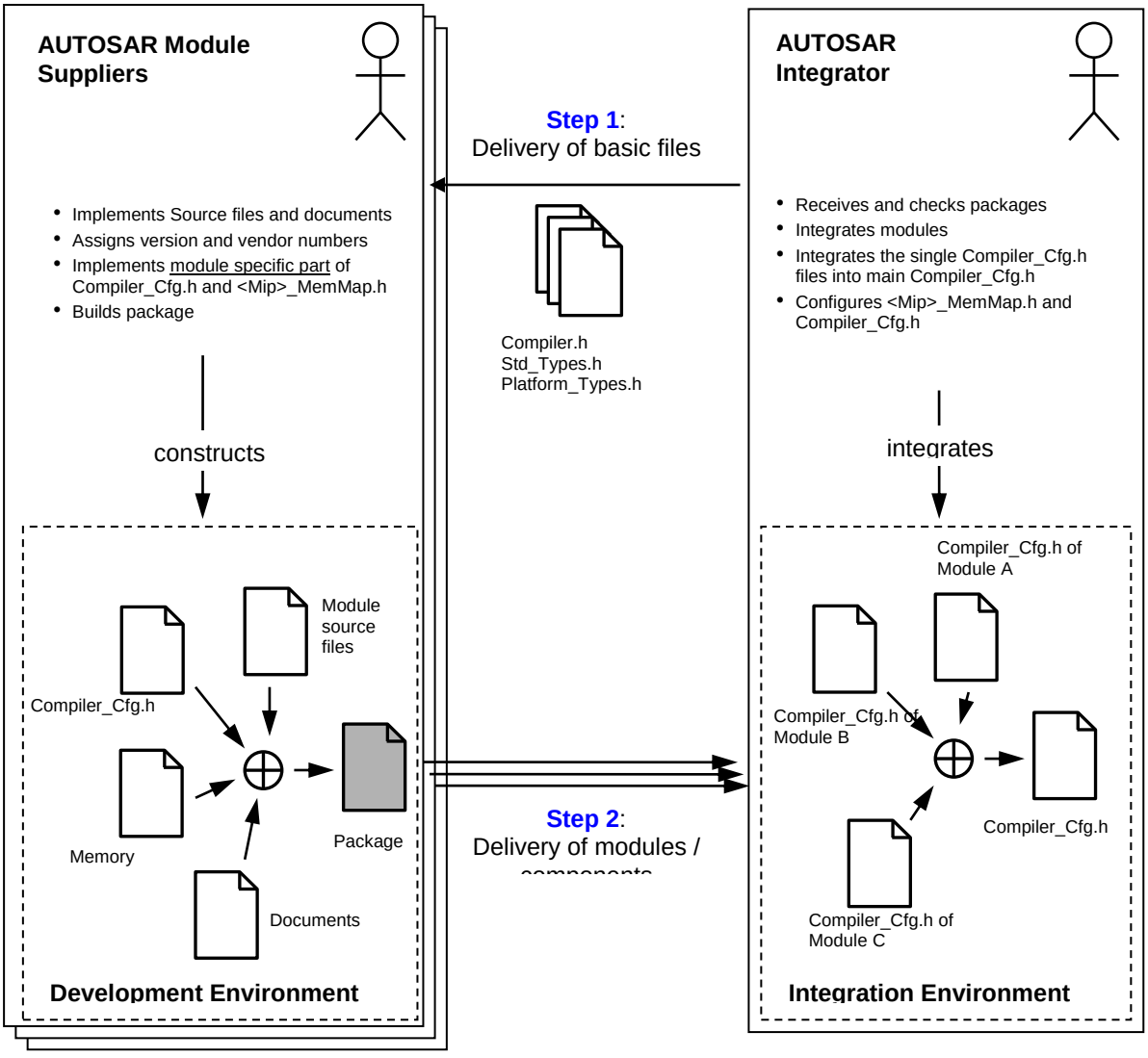

Figure 2: Proposal of integration-process

7.2 Development Errors

Not applicable.

7.3 Production Errors

Not applicable.

7.4 Extended Production Errors

Not applicable.

7.5 Error detection

Not applicable.

7.6 Error notification

Not applicable.

7.7 Version check

Not applicable.

7.8 Support for Debugging

Not applicable.

8 API specification

8.1 Imported types

Not applicable.

8.2 Macro definitions

These kind of items are the only API applicable to this module.

8.2.1 General definitions

8.2.1.1 Memory class AUTOMATIC

[SWS_COMPILER_00046][

Define:	AUTOMATIC
Range:	"empty" --
Description:	The memory class AUTOMATIC shall be provided as empty definition, used for the declaration of local pointers.
Caveats:	SWS_COMPILER_00040

] (SRS_BSW_00361)

8.2.1.2 Memory class TYPEDEF

[SWS_COMPILER_00059][

Define:	TYPEDEF
Range:	"empty" --
Description:	The memory class TYPEDEF shall be provided as empty definition. This memory class shall be used within type definitions, where no memory qualifier can be specified. This can be necessary for defining pointer types, with e.g. P2VAR, where the macros require two parameters. First parameter can be specified in the type definition (distance to the memory location referenced by the pointer), but the second one (memory allocation of the pointer itself) cannot be defined at this time. Hence, memory class TYPEDEF shall be applied.
Caveats:	SWS_COMPILER_00040

] (SRS_BSW_00404)

8.2.1.3 NULL_PTR
[SWS_COMPILER_00051][

Define:	NULL_PTR
Range:	void pointer `((void *)0)`
Description:	The compiler abstraction shall provide the NULL_PTR define with a void pointer to zero definition.
Caveats:	SWS_COMPILER_00040

] (SRS_BSW_00480)

8.2.1.4 INLINE
[SWS_COMPILER_00057][

Define:	INLINE
Range:	inline/"empty" --
Description:	The compiler abstraction shall provide the INLINE define for abstraction of the keyword inline.
Caveats:	SWS_COMPILER_00040

] (SRS_BSW_00361)

8.2.1.5 LOCAL_INLINE
[SWS_COMPILER_00060][

Define:	LOCAL_INLINE
Range:	static inline/"empty" --
Description:	The compiler abstraction shall provide the LOCAL_INLINE define for abstraction of the keyword inline in functions with "static" scope.
Caveats:	Different compilers may require a different sequence of the keywords "static" and "inline" if this is supported at all.

] (SRS_BSW_00361)

8.2.2 Function definitions

The following tables do not contain requirements. They just give an overview of used function keywords and their syntax within different compilers. This analysis is required for a correct and complete specification of methods and keywords and as rationale for those people who doubt the necessity of a compiler abstraction in AUTOSAR. These tables are not the complete overview of all existing compilers and platforms and their usage in AUTOSAR. However, the tables show examples that cover most use cases, from which the concepts are derived.

On platforms with memory exceeding the addressable range of the architecture (e.g. S12X with 512k of Flash) the compiler needs to know if a called function is reachable within normal addressing commands ('near') or extended/paged addressing commands ('far').

Compiler analysis for near functions:

Compiler	***Required syntax***
Cosmic, S12X	`@near void MyNearFunction(void);` Call of a near function results in a local page call or to a call into direct page. Dependent of compiler settings the compiler controls only the calling convention or allocation and calling convention.
Metrowerks, S12X	`void __near MyNearFunction(void);` Call of a near function results in a local page call or to a call into direct page.
IAR, HCS12 C/C++	`void __non_banked MyNearFunction (void);`

Tasking, ST10	`void _near MyNearFunction (void);`
	`_near void MyNearFunction (void);`
	Call of a near function results in a local segment code access (relevant in large model).

Tasking, TC1796	`void MyNearFunction (void);` (No keywords required)
Greenhills, V850	`void MyNearFunction (void);` (No keywords required)
ADS, ST30	`void MyNearFunction (void);` (No keywords required)
DIABDATA, MPC5554	`void MyNearFunction (void);` (No keywords required)

Compiler analysis for far functions:

Compiler	***Required syntax***
Cosmic, S12X	`@far void MyFarFunction(void);` Dependent of compiler settings the compiler controls only the calling convention or allocation and calling convention.
Metrowerks, S12X	`void __far MyFarFunction(void);`
IAR, HCS12 C/C++	`void __banked MyFarFunction (void);`
Tasking, ST10	`void _huge MyFarFunction (void);` `_huge void MyFarFunction (void);`
Tasking, TC1796	`void MyFarFunction (void);` (No keywords required)
Greenhills, V850	`void MyFarFunction (void);` (No keywords required)
ADS, ST30	`void MyFarFunction (void);` (No keywords required)
DIABDATA, MPC5554	`void MyFarFunction (void);` (No keywords required)

8.2.2.1 FUNC identification information

[SWS_COMPILER_00001][

Macro name:	FUNC	
Syntax:	`#define FUNC(rettype, memclass)`	
Parameters (in):	`retype`	return type of the function
	`memclass`	classification of the function itself
Parameters (out):	`None`	--
Return value:	`None`	--
Description:	The compiler abstraction shall define the FUNC macro for the declaration and definition of functions that ensures correct syntax of function declarations as required by a specific compiler.	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00003)

Example (Cosmic, S12X):

```
#define <PREFIX>_CODE          @near
#define FUNC(rettype, memclass) memclass rettype
```

Required usage for function declaration and definition:

```
FUNC(void, <PREFIX>_CODE) ExampleFunction (void);
```

[SWS_COMPILER_00058][In the parameter list of this macro no further Compiler Abstraction macros shall be nested. Instead, use a previously defined type as return type or use FUNC_P2CONST/FUNC_P2VAR.
] (SRS_BSW_00306)

Example:

```
 typedef P2VAR(uint8, AUTOMATIC, <PREFIX>_VAR) NearDataType;
 FUNC(NearDataType, <PREFIX>_CODE)
FarFuncReturnsNearPtr(void);
```

8.2.2.2 FUNC_P2CONST

[SWS_COMPILER_00061][

Macro name:	FUNC_P2CONST	
Syntax:	`#define FUNC_P2CONST(rettype, ptrclass, memclass)`	
Parameters (in):	`rettype`	return type of the function
	`ptrclass`	defines the classification of the pointer's distance
	`memclass`	classification of the function itself
Parameters (out):	`none`	--
Return value:	`none`	--
Description:	The compiler abstraction shall define the FUNC_P2CONST macro for the declaration and definition of functions returning a pointer to a constant. This shall ensure the correct syntax of function declarations as required by a specific compiler.	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Cosmic, S12X):

```
#define <PREFIX>_PBCFG        @far
#define <PREFIX>_CODE         @near
#define FUNC_P2CONST(rettype, ptrclass, memclass)\
const ptrclass rettype * memclass
```

Required usage for function declaration and definition:

```
FUNC_P2CONST(uint16, <PREFIX>_PBCFG, <PREFIX>_CODE)
ExampleFunction (void);
```

[SWS_COMPILER_00062][In the parameter list of the FUNC_P2CONST, no further Compiler Abstraction macros shall be nested.
] (SRS_BSW_00361)

8.2.2.3 FUNC_P2VAR

[SWS_COMPILER_00063][

Macro name:	FUNC_P2VAR	
Syntax:	`#define FUNC_P2VAR(rettype, ptrclass, memclass)`	
Parameters (in):	`rettype`	return type of the function
	`ptrclass`	defines the classification of the pointer's distance
	`memclass`	classification of the function itself
Parameters (out):	`none`	--
Return value:	`none`	--
Description:	The compiler abstraction shall define the FUNC_P2VAR macro for the declaration and definition of functions returning a pointer to a variable. This shall ensure the correct syntax of function declarations as required by a specific compiler.	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Cosmic, S12X):

```
#define <PREFIX>_PBCFG        @far
#define <PREFIX>_CODE         @near
#define FUNC_P2VAR(rettype, ptrclass, memclass)\
ptrclass rettype * memclass
```

Required usage for function declaration and definition:

```
FUNC_P2VAR(uint16, <PREFIX>_PBCFG, <PREFIX>_CODE)
ExampleFunction (void);
```

[SWS_COMPILER_00064][In the parameter list of the macro FUNC_P2VAR, no further Compiler Abstraction macros shall be nested.
] (SRS_BSW_00361)

8.2.3 Pointer definitions

The following tables do not contain requirements. They just give an overview of used pointer keywords and their syntax within different compilers. This analysis is required for a correct and complete specification of methods and keywords and as rationale for those people who doubt the necessity of a compiler abstraction in AUTOSAR. These tables are not the complete overview of all existing compilers and platforms and their usage in AUTOSAR. However, the tables show examples that cover most use cases, from which the concepts are derived.

On platforms with memory exceeding the addressable range of the architecture (e.g. S12X with 512k of Flash) the compiler needs to know if data referenced by a pointer is accessible by normal addressing commands ('near') or extended/paged addressing commands ('far').

Compiler analysis for near pointers pointing to variable_data in RAM (use case: pointer to data buffer where data has to be copied to):

Compiler	***Required syntax***
Cosmic, S12X	`@near uint8* MyNearPointer;`
Metrowerks, S12X	`uint8* __near MyNearPointer;`
IAR, HCS12 C/C++	`uint8* __data16 MyNearPointer;`
Tasking, ST10	`_near uint8* MyNearPointer;`
Tasking, TC1796	`uint8* MyNearPointer;` (No keywords required)
Greenhills, V850	`uint8* MyNearPointer` (No keywords required)
ADS, ST30	`uint8* MyNearPointer` (No keywords required)
DIABDATA, MPC5554	`uint8* MyNearPointer` (No keywords required)

Compiler analysis for far pointers pointing to variable data in RAM:

Compiler	***Required syntax***
Cosmic, S12X	`@far uint8* MyFarPointer;`
Metrowerks, S12X	`uint8* __far MyFarPointer;`
IAR, HCS12 C/C++	(Information not available yet)

Tasking, ST10	`_far uint8* MyFarPointer;  /*14 bit arithmetic*/`
	`_huge uint8* MyFarPointer;  /*24 bit arithmetic*/`
	`_shuge uint8* MyFarPointer; /*16 bit arithmetic*/`
	`/* My personal note: CRAZY */`

Tasking, TC1796	`uint8* MyFarPointer;` (No keywords required)
Greenhills, V850	`uint8* MyFarPointer` (No keywords required)
ADS, ST30	`uint8* MyFarPointer` (No keywords required)
DIABDATA, MPC5554	`uint8* MyFarPointer` (No keywords required)

Compiler analysis for near pointers pointing to constant data in RAM (use case pointer to data buffer where data has to be read from):

Compiler	***Required syntax***
Cosmic, S12X	`@near uint8* MyNearPointer;` (Results in access of direct memory area)
Metrowerks, S12X	`const uint8* __near MyNearPointer;` (Results in access of direct memory area)
IAR, HCS12 C/C++	`const uint8*  MyNearPointer;` (Results in access of direct memory area)
Tasking, ST10	`const _near uint8* MyNearPointer;`
Tasking, TC1796	`const _near uint8* MyNearPointer;`
Greenhills, V850	`const uint8* MyNearPointer` (No additional keywords required)
ADS, ST30	`const uint8* MyNearPointer` (No additional keywords required)
DIABDATA, MPC5554	`const uint8* MyNearPointer` (No additional keywords required)

Compiler analysis for far pointers pointing to constant data in RAM:

Compiler	***Required syntax***
Cosmic, S12X	`@far uint8* MyFarPointer;`
Metrowerks, S12X	`const uint8* __far MyFarPointer;`
IAR, HCS12 C/C++	(Information not available yet)
Tasking, ST10	`const _far uint8* MyFarPointer;`
Tasking, TC1796	`uint8* MyFarPointer;` (No keywords required)
Greenhills, V850	`const uint8* MyFarPointer` (No additional keywords required)
ADS, ST30	`const uint8* MyFarPointer` (No additional keywords required)
DIABDATA, MPC5554	`const uint8* MyFarPointer` (No additional keywords required)

Compiler analysis for near pointers pointing to data in ROM (use case pointer to display data in ROM passed to SPI Driver):

Compiler	***Required syntax***
Cosmic, S12X	`const uint8* MyNearPointer;` `(Without near keyword because this is by default near!)`
Metrowerks, S12X	`const uint8* __near MyNearPointer;`
IAR, HCS12 C/C++	`const uint8* MyNearPointer;` `(Without near keyword because this is by default near!)`
Tasking, ST10	`const _near uint8* MyNearPointer;`
Tasking, TC1796	`const uint8* MyNearPointer;` (No keywords required)
Greenhills, V850	`const uint8* MyNearPointer` (No additional keywords required)
ADS, ST30	`const uint8* MyNearPointer` (No additional keywords required)
DIABDATA, MPC5554	`const uint8* MyNearPointer` (No additional keywords required)

Compiler analysis for far pointers pointing to constant data in ROM:

Compiler	***Required syntax***
Cosmic, S12X	not possible
Metrowerks, S12X	`const uint8* __far MyFarPointer;`
IAR, HCS12 C/C++	Access function and the banked constant data are located in the same bank: `const uint8* MyFarPointer;` but caller shall use the `__address_24_of` macro Access function is located in non-banked memory: PPAGE register has to be handled manually Access function and the banked constant data are located in different banks: Not possible
Tasking, ST10	`const _far uint8* MyFarPointer;`
Tasking, TC1796	`const uint8* MyFarPointer;` (No keywords required)
Greenhills, V850	`const uint8* MyFarPointer` (No additional keywords required)
ADS, ST30	`const uint8* MyFarPointer` (No additional keywords required)
DIABDATA, MPC5554	`const uint8* MyFarPointer` (No additional keywords required)

The HW architecture of the S12X supports different paging mechanisms with different limitations e.g. supported instruction set or pointer distance. Therefore the IAR, HCS12 C/C++ and the Cosmic, S12X compilers are limited in the usage of generic pointers applicable for the whole memory area because of the expected code overhead.

Conclusion: These vendors should adapt their compilers, because a generic SW architecture as described by AUTOSAR cannot be adjusted in every case to the platform specific optimal solution.

Compiler analysis for pointers, where the symbol of the pointer itself is placed in near-memory:

Compiler	***Required syntax***
Cosmic, S12X	`uint8* @near MyPointerInNear;`
Metrowerks, S12X	`__near uint8* MyPointerInNear;`
Tasking, ST10	`uint8* _near MyPointerInNear;`
Tasking, TC1796	`uint8* MyPointerInNear;` (No keywords required)
Greenhills, V850	`uint8* MyPointerInNear` (No keywords required)
ADS, ST30	`uint8* MyPointerInNear` (No keywords required)
DIABDATA, MPC5554	`uint8* MyPointerInNear` (No keywords required)

Compiler analysis for pointers, where the symbol of the pointer itself is placed in far-memory:

Compiler	***Required syntax***
Cosmic, S12X	`uint8* @far MyPointerInFar;`
Metrowerks, S12X	`__far uint8* MyPointerInFar;`
Tasking, ST10	`uint8* _far MyPointerInFar;`
Tasking, TC1796	`uint8* MyPointerInFar;` (No keywords required)
Greenhills, V850	`uint8* MyPointerInFar` (No keywords required)
ADS, ST30	`uint8* MyPointerInFar` (No keywords required)
DIABDATA, MPC5554	`uint8* MyPointerInFar` (No keywords required)

The examples above lead to the conclusion, that for definition of a pointer it is not sufficient to specify only one memory class. Instead, a combination of two memory classes, one for the pointer's 'distance' and one for the pointer's symbol itself, is possible, e.g.:

```
/* Tasking ST10, far-pointer in near memory
 * (both content and pointer in RAM)
 */
_far uint8* _near MyFarPointerInNear;
```

Compiler analysis for function pointers:

Compiler	***Required syntax***
Cosmic, S12X	`@near void (* const Irq_InterruptVectorTable[])(void)` Call of a near function results in an interpage call or to a call into direct page:
Metrowerks, S12X	`void (*const __near Irq_InterruptVectorTable[]) (void)` Call of a near function results in an interpage call or to a call into direct page: Near functions and far functions are not compatible because of other ret-statements:
IAR, HCS12 C/C++	`__non_banked void (* const Irq_InterruptVectorTable[])(void)` Casting from __non_banked to __banked is performed through zero extension: Casting from __banked to __non_banked is an illegal operation.
Tasking, ST10	`_far void (*NvM_AsyncCbkPtrType) (NvM_ModuleIdType ModuleId, NvM_ServiceIdType ServiceId )` Call of a near function results in a local segment code access (relevant in large model):
Tasking, TC1796	`void (*NvM_AsyncCbkPtrType) (NvM_ModuleIdType ModuleId, NvM_ServiceIdType ServiceId )` (No additional keywords required)
Greenhills, V850	`void (*NvM_AsyncCbkPtrType) (NvM_ModuleIdType ModuleId,`

Compiler	Required syntax
	NvM_ServiceIdType ServiceId) (No additional keywords required)
ADS, ST30	void (*NvM_AsyncCbkPtrType) (NvM_ModuleIdType ModuleId, NvM_ServiceIdType ServiceId) (No additional keywords required)
DIABDATA, MPC5554	void (*NvM_AsyncCbkPtrType) (NvM_ModuleIdType ModuleId, NvM_ServiceIdType ServiceId) (No additional keywords required)

8.2.3.1 P2VAR

[SWS_COMPILER_00006][

Macro name:	P2VAR	
Syntax:	`#define P2VAR(ptrtype, memclass, ptrclass)`	
Parameters (in):	`ptrtype`	type of the referenced variable
	`memclass`	classification of the pointer's variable itself
	`ptrclass`	defines the classification of the pointer's distance
Parameters (out):	`none`	--
Return value:	`none`	--
Description:	The compiler abstraction shall define the P2VAR macro for the declaration and definition of pointers in RAM, pointing to variables. The pointer itself is modifiable (e.g. ExamplePtr++). The pointer's target is modifiable (e.g. *ExamplePtr = 5).	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Metrowerks, S12X):

```
#define P2VAR(ptrtype, memclass, ptrclass) \
                ptrclass ptrtype * memclass
```

Required usage for pointer declaration and definition:

```
#define SPI_APPL_DATA @far
#define SPI_VAR_FAST  @near

P2VAR(uint8, SPI_VAR_FAST, SPI_APPL_DATA) Spi_FastPointerToApplData;
```

8.2.3.2 P2CONST

[SWS_COMPILER_00013][

Macro name:	P2CONST	
Syntax:	`#define P2CONST(ptrtype, memclass, ptrclass)`	
Parameters (in):	`ptrtype`	type of the referenced constant
	`memclass`	classification of the pointer's variable itself
	`ptrclass`	defines the classification of the pointer's distance
Parameters (out):	`none`	--
Return value:	`none`	--
Description:	The compiler abstraction shall define the P2CONST macro for the declaration and definition of pointers in RAM pointing to constants The pointer itself is modifiable (e.g. ExamplePtr++). The pointer's target is not modifiable (read only).	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Metrowerks, S12X):

```
#define P2CONST(ptrtype, memclass, ptrclass) \
                const ptrtype memclass * ptrclass
```

Example (Cosmic, S12X):

```
#define P2CONST(ptrtype, memclass, ptrclass) \
                const ptrtype ptrclass * memclass
```

Example (Tasking, ST10):

```
#define P2CONST(ptrtype, memclass, ptrclass) \
                const ptrclass ptrtype * memclass
```

Required usage for pointer declaration and definition:

```
#define EEP_APPL_CONST @far
#define EEP_VAR  @near

P2CONST(Eep_ConfigType, EEP_VAR, EEP_APPL_CONST) Eep_ConfigurationPtr;
```

8.2.3.3 CONSTP2VAR
[SWS_COMPILER_00031][

Macro name:	CONSTP2VAR	
Syntax:	`#define CONSTP2VAR (ptrtype, memclass, ptrclass)`	
Parameters (in):	`ptrtype`	type of the referenced variable
	`memclass`	classification of the pointer's constant itself
	`ptrclass`	defines the classification of the pointer's distance
Parameters (out):	`None`	--
Return value:	`None`	--
Description:	The compiler abstraction shall define the CONSTP2VAR macro for the declaration and definition of constant pointers accessing variables. The pointer itself is not modifiable (fix address). The pointer's target is modifiable (e.g. *ExamplePtr = 18).	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Tasking, ST10):
```
#define CONSTP2VAR (ptrtype, memclass, ptrclass) \
                ptrclass ptrtype * const memclass
```

Required usage for pointer declaration and definition:
```
/* constant pointer to application data */
CONSTP2VAR (uint8, NVM_VAR, NVM_APPL_DATA)
NvM_PointerToRamMirror = Appl_RamMirror;
```

8.2.3.4 CONSTP2CONST
[SWS_COMPILER_00032][

Macro name:	CONSTP2CONST	
Syntax:	`#define CONSTP2CONST(ptrtype, memclass, ptrclass)`	
Parameters (in):	`ptrtype`	type of the referenced constant
	`memclass`	classification of the pointer's constant itself
	`ptrclass`	defines the classification of the pointer's distance
Parameters (out):	`none`	--
Return value:	`none`	--
Description:	The compiler abstraction shall define the CONSTP2CONST macro for the declaration and definition of constant pointers accessing constants. The pointer itself is not modifiable (fix address). The pointer's target is not modifiable (read only).	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Tasking, ST10):
```
#define  CONSTP2CONST (ptrtype, memclass, ptrclass) \
                const memclass ptrtype * const ptrclass
```

Required usage for pointer declaration and definition:
```
#define CAN_PBCFG_CONST @gpage
#define CAN_CONST       @near

/* constant pointer to the constant postbuild configuration
data */
CONSTP2CONST (Can_PBCfgType, CAN_CONST, CAN_PBCFG_CONST)
Can_PostbuildCfgData = CanPBCfgDataSet;
```

8.2.3.5 P2FUNC

[SWS_COMPILER_00039][

Macro name:	P2FUNC	
Syntax:	`#define P2FUNC(rettype, ptrclass, fctname)`	
Parameters (in):	`rettype`	return type of the function
	`ptrclass`	defines the classification of the pointer's distance
	`fctname`	function name respectively name of the defined type
Parameters (out):	`None`	--
Return value:	`None`	--
Description:	The compiler abstraction shall define the P2FUNC macro for the type definition of pointers to functions.	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Metrowerks, S12X):
```
define P2FUNC(rettype, ptrclass, fctname)\
                rettype (*ptrclass fctname)
```

Example (Cosmic, S12X):
```
#define P2FUNC(rettype, ptrclass, fctname) \
                ptrclass rettype (*fctname)
```

Required usage for pointer type declaration:
```
#define EEP_APPL_CONST @far
#define EEP_VAR  @near

typedef P2FUNC (void, NVM_APPL_CODE, NvM_CbkFncPtrType)
(void);
```

8.2.3.6 CONSTP2FUNC

[SWS_COMPILER_00065]⌈

Macro name:	CONSTP2FUNC	
Syntax:	`#define CONSTP2FUNC(rettype, ptrclass, fctname)`	
Parameters (in):	`rettype`	return type of the function
	`ptrclass`	defines the classification of the pointer's distance
	`fctname`	function name respectively name of the defined type
Parameters (out):	`None`	--
Return value:	`None`	--
Description:	The compiler abstraction shall define the CONSTP2FUNC macro for the type definition of constant pointers to functions.	
Caveats:	--	
Configuration:	--	

⌋ (SRS_BSW_00361)

Example (PowerPC):

```
#define CONSTP2FUNC(rettype, ptrclass, fctname)\
                    rettype (* const fctname)
```

Example (CodeWarrior, S12X):

```
#define CONSTP2FUNC(rettype, ptrclass, fctname)\
                    rettype (* const ptrclass fctname)
```

8.2.4 Constant definitions

8.2.4.1 CONST

[SWS_COMPILER_00023]⌈

Macro name:	CONST	
Syntax:	`#define CONST(consttype, memclass)`	
Parameters (in):	`consttype`	type of the constant
	`memclass`	classification of the constant itself
Parameters (out):	`none`	--
Return value:	`none`	--
Description:	The compiler abstraction shall define the CONST macro for the declaration and definition of constants.	
Caveats:	--	
Configuration:	--	

⌋ (SRS_BSW_00309)

Example (Cosmic, S12X):

```
#define CONST(type, memclass) memclass const type
```

Required usage for declaration and definition:

```
#define NVM_CONST @gpage

CONST(uint8, NVM_CONST) NvM_ConfigurationData;
```

8.2.5 Variable definitions

8.2.5.1 VAR [SWS_COMPILER_00026][

Macro name:	VAR	
Syntax:	`#define VAR(vartype, memclass)`	
Parameters (in):	`vartype`	type of the variable
	`memclass`	classification of the variable itself
Parameters (out):	`None`	--
Return value:	`None`	--
Description:	The compiler abstraction shall define the VAR macro for the declaration and definition of variables.	
Caveats:	--	
Configuration:	--	

] (SRS_BSW_00361)

Example (Tasking, ST10):

```
#define VAR(type, memclass) memclass type
```

Required usage for declaration and definition:

```
#define NVM_FAST_VAR _near

VAR(uint8, NVM_FAST_VAR) NvM_VeryFrequentlyUsedState;
```

8.3 Type definitions

Not applicable.

8.4 Function definitions

Not applicable.

8.5 Call-back notifications

Not applicable.

8.6 Scheduled functions

Not applicable.

8.7 Expected Interfaces

8.7.1 Mandatory Interfaces

Not applicable.

8.7.2 Optional Interfaces

Not applicable.

8.7.3 Configurable interfaces

Not applicable.

8.8 Service Interfaces

8.8.1 Scope of this Chapter

Not applicable.

8.8.2 Overview

Not applicable.

8.8.3 Specification of the Ports and Port Interfaces

8.8.3.1 General Approach

Not applicable.

8.8.3.2 Data Types

Not applicable.

8.8.3.3 Port Interface

Not applicable.

8.8.4 Definition of the Service

Not applicable.

8.8.5 Configuration of the DET

Not applicable.

9 Sequence diagrams

Not applicable.

10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification, Chapter 10.1 describes fundamentals. We intend to leave Chapter 10.1 in the specification to guarantee comprehension.⌋ (SRS_BSW_00389)

Chapter 10.2 specifies the structure (containers) and the parameters of this module.

Chapter 10.3 specifies published information of this module.

The Compiler Abstraction has no separate configuration interface by means of specifying a separate parameter definition. Instead, configuration of the Memory Mapping has been extended (see [13]) by the parameters described in this chapter.

10.1 How to read this chapter

In addition to this section, it is highly recommended to read the documents:
Layered Software Architecture [3]
Specification Of ECU Configuration [4]

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

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe Chapters 8 and Chapter 9.

10.2.1 Module-Specific Memory Classes

It is also possible to configure module-specific memory classes. This is done by using the container 'MemMapAddressingModeSet' and the contained parameter 'MemMapCompilerMemClassSymbolImpl'. For detailed information about these configuration parameters refer to [13].

10.2.2 Global Memory Classes

Furthermore it is possible to configure global memory classes that are valid for all modules. This is done by using the container 'MemMapGenericCompilerMemClass' and the contained parameter 'MemMapGenericCompilerMemClassSymbolImpl'. For detailed information about these configuration parameters refer to [13].
[SWS_COMPILER_00042] ⌈ The file Compiler.h is specific for each build scenario.

Therefore there is no standardized configuration interface specified.⌋ (SRS_BSW_00361)

10.3 Published Information

Not applicable.

11 Not applicable requirements

[SWS_COMPILER_00999] [These requirements are not applicable to this specification. **]** (SRS_BSW_00300, SRS_BSW_00301, SRS_BSW_00302, SRS_BSW_00305, SRS_BSW_00307, SRS_BSW_00308, SRS_BSW_00309, SRS_BSW_00310, SRS_BSW_00312, SRS_BSW_00314, SRS_BSW_00323, SRS_BSW_00325, SRS_BSW_00327, SRS_BSW_00330, SRS_BSW_00331, SRS_BSW_00333, SRS_BSW_00334, SRS_BSW_00335, SRS_BSW_00336, SRS_BSW_00339, SRS_BSW_00341, SRS_BSW_00342, SRS_BSW_00343, SRS_BSW_00344, SRS_BSW_00346, SRS_BSW_00350, SRS_BSW_00353, SRS_BSW_00357, SRS_BSW_00358, SRS_BSW_00359, SRS_BSW_00360, SRS_BSW_00369, SRS_BSW_00371, SRS_BSW_00373, SRS_BSW_00375, SRS_BSW_00377, SRS_BSW_00378, SRS_BSW_00380, SRS_BSW_00385, SRS_BSW_00386, SRS_BSW_00390, SRS_BSW_00392, SRS_BSW_00393, SRS_BSW_00394, SRS_BSW_00395, SRS_BSW_00398, SRS_BSW_00399, SRS_BSW_00004, SRS_BSW_00400, SRS_BSW_00401, SRS_BSW_00404, SRS_BSW_00405, SRS_BSW_00406, SRS_BSW_00407, SRS_BSW_00408, SRS_BSW_00409, SRS_BSW_00410, SRS_BSW_00411, SRS_BSW_00413, SRS_BSW_00414, SRS_BSW_00415, SRS_BSW_00416, SRS_BSW_00417, SRS_BSW_00419, SRS_BSW_00422, SRS_BSW_00423, SRS_BSW_00424, SRS_BSW_00425, SRS_BSW_00426, SRS_BSW_00427, SRS_BSW_00428, SRS_BSW_00429, SRS_BSW_00432, SRS_BSW_00433, SRS_BSW_00005, SRS_BSW_00007, SRS_BSW_00009, SRS_BSW_00010, SRS_BSW_00158, SRS_BSW_00161, SRS_BSW_00162, SRS_BSW_00164, SRS_BSW_00167, SRS_BSW_00168, SRS_BSW_00170, SRS_BSW_00171, SRS_BSW_00172, SRS_BSW_00351, SRS_BSW_00383, SRS_BSW_00388, SRS_BSW_00389, SRS_BSW_00396, SRS_BSW_00403, SRS_BSW_00412, SRS_BSW_00448, SRS_BSW_00449, SRS_BSW_00452, SRS_BSW_00453, SRS_BSW_00454, SRS_BSW_00456, SRS_BSW_00457, SRS_BSW_00458, SRS_BSW_00459, SRS_BSW_00461, SRS_BSW_00462, SRS_BSW_00466, SRS_BSW_00469, SRS_BSW_00470, SRS_BSW_00471, SRS_BSW_00472, SRS_BSW_00473, SRS_BSW_00478, SRS_BSW_00479, SRS_BSW_00480, SRS_BSW_00481, SRS_BSW_00482, SRS_BSW_00483)