Document Title	Specification of Compiler Abstraction
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	051
Document Classification	Standard

Document Version	3.0.0
Document Status	Final
Part of Release	4.0
Revision	1

Document Change History

Date	Version	Changed by	Change Description
02.12.2009	3.0.0	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
23.06.2008	2.0.1	AUTOSAR Administration	• Legal disclaimer revised
27.11.2007	2.0.0	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
31.01.2007	1.1.0	AUTOSAR Administration	• Add: COMPILER058 • Add: COMPILER057 • Change: COMPILER040 • Legal disclaimer revised • Release Notes added • “Advice for users” revised • “Revision Information” added
27.04.2006	1.0.0	AUTOSAR Administration	Initial Release

Disclaimer

Advice for users

AUTOSAR specifications 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 specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, 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 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 (e.g. EEP_CODE).

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).
C89	ANSI X3.159-1989 Programming Language C
C90	ISO/IEC 9899:1990
C99	ISO/IEC 9899:1999, 2nd edition, 1. December 1999
EmbeddedC	ISO/IEC DTR 18037, draft standard, 24. September 2003

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

3.2 Related standards and norms

[12] ANSI X3.159-1989 Programming Language C

[13] ISO/IEC 9899:1990

[14] ISO/IEC 9899:1999, 2nd edition, 1. December 1999

[15] ISO/IEC DTR 18037, draft standard, 24. September 2003

4 Constraints and assumptions

4.1 Limitations

During specification of abstraction and validation of concept the compilers listed in chapter 3.1 have 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.

The concepts described in this document do only apply to C compilers according the standard C90. C++ is not in scope of this version.
In contradiction to the C-standard, some extensions are required:

- keywords for interrupt declaration
- keywords for hardware specific memory modifier
- uninitialized variables

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 reused. The C language (and other languages as well) can not 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

COMPILER048: 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 COMPILER040) shall fulfill the implementation and configuration specific needs of each software module in a specific build scenario.

5.1 Code file structure

Not applicable

5.2 Header file structure

COMPILER052: Include structure of the compiler specific language extension header:

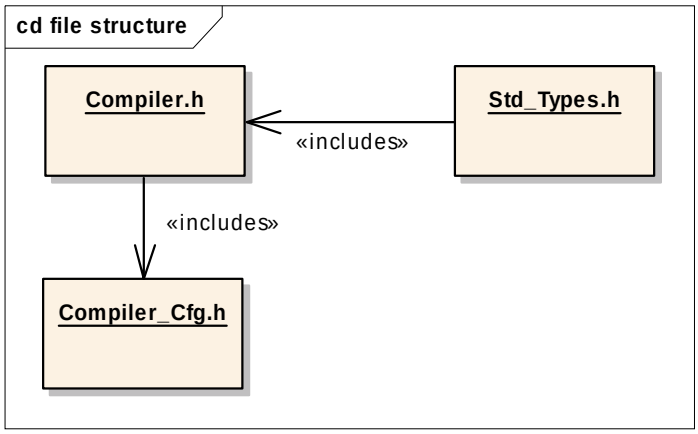

Figure 1: Include structure of Compiler.h

5.3 Connections to other modules

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

5.3.1 Compiler Abstraction

As described in this document, the compiler abstraction is used to configure the reachability of elements (pointers, variables, function etc.)

5.3.2 Memory Mapping

This module is used to do the sectioning of memory. The user can define sections for optimizing the source code.

5.3.3 Linker-Settings

The classification which elements are assigned to which memory section can be done by linker-settings.

6 Requirements traceability

Document: AUTOSAR requirements on Basic Software, general

Requirement	***Satisfied by***
[BSW003] Version identification	COMPILER001_PI
[BSW00300] Module naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00301] Limit imported information	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00302] Limit exported information	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00304] AUTOSAR integer data types	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00305] Self-defined data types naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00306] Avoid direct use of compiler and platform specific keywords	supported by: COMPILER001, COMPILER006, COMPILER010, COMPILER012, COMPILER013, COMPILER015, COMPILER023, COMPILER026, COMPILER031, COMPILER032, COMPILER033, COMPILER035, COMPILER036, COMPILER039, COMPILER044, COMPILER046
[BSW00307] Global variables naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00308] Definition of global data	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00309] Global data with read-only constraint	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00310] API naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00312] Shared code shall be reentrant	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00314] Separation of interrupt frames and service routines	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00318] Format of module version numbers	COMPILER001_PI
[BSW00321] Enumeration of module version numbers	COMPILER001_PI
[BSW00323] API parameter checking	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00324] Do not use HIS I/O Library	Not applicable (non-functional requirement)
[BSW00325] Runtime of interrupt service routines	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00326] Transition from ISRs to OS tasks	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00327] Error values naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00328] Avoid duplication of code	supported by: COMPILER048
[BSW00329] Avoidance of generic interfaces	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00330] Usage of macros / inline functions instead of functions	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00331] Separation of error and status values	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00333] Documentation of callback function context	Not applicable (Compiler Abstraction is not a BSW module)

Requirement	*Satisfied by*
[BSW00334] Provision of XML file	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00335] Status values naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00336] Shutdown interface	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00337] Classification of errors	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00338] Detection and Reporting of development errors	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00339] Reporting of production relevant error status	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00341] Microcontroller compatibility documentation	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00342] Usage of source code and object code	Not applicable (non-functional requirement)
[BSW00343] Specification and configuration of time	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00344] Reference to link-time configuration	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00345] Pre-compile-time configuration	Chapter 11.2.1
[BSW00346] Basic set of module files	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00347] Naming separation of different instances of BSW drivers	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00348] Standard type header	COMPILER003, COMPILER004, COMPILER052
[BSW00350] Development error detection keyword	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00353] Platform specific type header	Not applicable (Compiler Abstraction is the C-language extension header)
[BSW00355] Do not redefine AUTOSAR integer data types	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00357] Standard API return type	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00358] Return type of init() functions	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00359] Return type of callback functions	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00360] Parameters of callback functions	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00361] Compiler specific language extension header	COMPILER003, COMPILER004
[BSW00369] Do not return development error codes via API	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00370] Separation of callback interface from API	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00371] Do not pass function pointers via API	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00373] Main processing function naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00374] Module vendor identification	COMPILER001_PI
[BSW00375] Notification of wake-up reason	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00376] Return type and parameters of main processing functions	Not applicable (Compiler Abstraction is not a BSW module)

Requirement	*Satisfied by*
[BSW00377] Module specific API return types	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00378] AUTOSAR boolean type	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00379] Module identification	COMPILER001_PI
[BSW00380] Separate C-Files for configuration parameters	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00381] Separate configuration header file for pre-compile time parameters	COMPILER052
[BSW00383] List dependencies of configuration files	Figure 1: Include structure of Compiler.h
[BSW00384] List dependencies to other modules	COMPILER048
[BSW00385] List possible error notifications	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00386] Configuration for detecting an error	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00387] Specify the configuration class of callback function	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00388] Introduce containers	Chapter 11.2
[BSW00389] Containers shall have names	**COMPILER044**
[BSW00390] Parameter content shall be unique within the module	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00391] Parameter shall have unique names	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00392] Parameters shall have a type	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00393] Parameters shall have a range	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00394] Specify the scope of the parameters	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00395] List the required parameters (per parameter)	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00396] Configuration classes	**COMPILER044**
[BSW00397] Pre-compile-time parameters	**COMPILER044**
[BSW00398] Link-time parameters	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00399] Loadable Post-build time parameters	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW004] Version check	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00400] Selectable Post-build time parameters	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00401] Documentation of multiple instances of configuration parameters	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00402] Published information	COMPILER001_PI
[BSW00404] Reference to post build time configuration	Not applicable (Compiler Abstraction is specific per build scenario)

Requirement	*Satisfied by*
[BSW00405] Reference to multiple configuration sets	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00406] Check module initialization	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00407] Function to read out published parameters	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00408] Configuration parameter naming convention	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00409] Header files for production code error IDs	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00410] Compiler switches shall have defined values	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00411] Get version info keyword	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00412] Separate H-File for configuration parameters	COMPILER052
[BSW00413] Accessing instances of BSW modules	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00414] Parameter of init function	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00415] User dependent include files	Not applicable (non-functional requirement)
[BSW00416] Sequence of Initialization	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00417] Reporting of Error Events by Non-Basic Software	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00419] Separate C-Files for pre-compile time configuration parameters	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW00420] Production relevant error event rate detection	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00421] Reporting of production relevant error events	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00422] Debouncing of production relevant error status	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00424] BSW main processing function task allocation	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00425] Trigger conditions for schedulable objects	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00426] Exclusive areas in BSW modules	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00427] ISR description for BSW modules	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00428] Execution order dependencies of main processing functions	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00429] Restricted BSW OS functionality access	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00431] The BSW Scheduler module implements task bodies	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path	Not applicable (Compiler Abstraction is not a BSW module)
[BSW00433] Calling of main processing functions	Not applicable (Compiler Abstraction is not a BSW module)

Requirement	*Satisfied by*
[BSW00434] The Schedule Module shall provide an API for exclusive areas	Not applicable (Compiler Abstraction is not a BSW module)
[BSW005] No hard coded horizontal interfaces within MCAL	Not applicable (non-functional requirement)
[BSW006] Platform independency	supported by: COMPILER001, COMPILER006, COMPILER010, COMPILER012, COMPILER013, COMPILER015, COMPILER023, COMPILER026, COMPILER031, COMPILER032, COMPILER033, COMPILER035, COMPILER036, COMPILER039, COMPILER044, COMPILER046
[BSW007] HIS MISRA C	Not applicable (Compiler Abstraction is the C-language extension header)
[BSW009] Module User Documentation	Not applicable (Compiler Abstraction is not a BSW module)
[BSW010] Memory resource documentation	Not applicable (Compiler Abstraction is not a BSW module)
[BSW101] Initialization interface	Not applicable (Compiler Abstraction is not a BSW module)
[BSW158] Separation of configuration from implementation	Not applicable (Compiler Abstraction is not a BSW module)
[BSW159] Tool-based configuration	Chapter 11.2.2
[BSW160] Human-readable configuration data	**COMPILER044**
[BSW161] Microcontroller abstraction	Not applicable (non-functional requirement)
[BSW162] ECU layout abstraction	Not applicable (non-functional requirement)
[BSW164] Implementation of interrupt service routines	Not applicable (non-functional requirement)
[BSW167] Static configuration checking	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW168] Diagnostic Interface of SW components	Not applicable (Compiler Abstraction is not a BSW module)
[BSW170] Data for reconfiguration of AUTOSAR SW-Components	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW171] Configurability of optional functionality	Not applicable (Compiler Abstraction is specific per build scenario)
[BSW172] Compatibility and documentation of scheduling strategy	Not applicable (Compiler Abstraction is not a BSW module)

7 Analysis

This chapter does not contain requirements. It just gives an overview of used 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. This chapter is no complete overview of existing compilers and platforms and their usage in AUTOSAR. But it shows examples that cover most use cases, from which the concepts specified in the consecutive chapters are derived.

7.1 Keywords for functions

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)

7.2 Keywords for pointers

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:

Compiler	***Required syntax***
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, 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 Functional specification

8.1 General issues

COMPILER003: For each compiler and platform an own compiler abstraction has to be provided.

8.2 Contents of Compiler.h

COMPILER004: The file name of the compiler abstraction shall be 'Compiler.h'.

COMPILER053: The file Compiler.h shall contain the definitions and macros specified in chapter 9. Those are fix for one specific compiler and platform.

COMPILER005: 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 */
```

COMPILER010: The compiler abstraction shall define a symbol for the target compiler according to the following naming convention:
_<COMPILERNAME>_C_<PLATFORMNAME>_

Note: 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

List of symbols: see COMPILER012

COMPILER030: "Compiler.h" shall provide information of the supported compiler vendor and the applicable compiler version.

COMPILER035: The macro parameters memclass and ptrclass shall not be filled with the compiler specific keywords but with one of the configured values in COMPILER040.

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.

COMPILER036: 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.

COMPILER047: 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.

COMPILER050: 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.

8.3 Contents of Compiler_Cfg.h

COMPILER055: 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 COMPILER040 for memory types and required syntax.

COMPILER054: 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 SoftwareComponent's name.

9 API specification

9.1 Definitions

9.1.1 memory class AUTOMATIC

Define:	AUTOMATIC
Range:	"empty" --
Description:	**COMPILER046**: The memory class AUTOMATIC shall be provided as empty definition, used for the declaration of local pointers.
Caveats:	COMPILER040

9.1.2 memory class TYPEDEF

Define:	TYPEDEF
Range:	"empty" --
Description:	**COMPILER059:** 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:	COMPILER040

9.1.3 NULL_PTR

Define:	NULL_PTR
Range:	void pointer `((void *)0)`
Description:	**COMPILER051**: The compiler abstraction shall provide the NULL_PTR define with a void pointer to zero definition.
Caveats:	--

9.1.4 INLINE

Define:	INLINE
Range:	inline/"empty" --
Description:	**COMPILER057**: The compiler abstraction shall provide the INLINE define for abstraction of the keyword inline.
Caveats:	--

9.1.5 LOCAL_INLINE

Define:	LOCAL_INLINE
Range:	static inline/"empty" --
Description:	**COMPILER060**: 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.

9.2 Macros for functions

9.2.1 FUNC

Macro name:	FUNC	
Syntax:	`#define FUNC(rettype, memclass)`	
Parameters (in):	`rettype`	return type of the function
	`memclass`	classification of the function itself
Parameters (out):	`none`	--
Return value:	`none`	--

Description:	**COMPILER001**: 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.
	
	COMPILER058: In the parameter list of this macro no further Compiler Abstraction macros shall be nested. Instead use a previously defined type as return type. Example:
	`typedef P2VAR(uint8, AUTOMATIC, _near) NearDataType;`
	`FUNC(NearDataType, _far) FarFuncReturnsNearPtr(void);`	

Caveats:	--	
Configuration:	--	

Example (Cosmic, S12X):
```
#define FUNC(rettype, memclass) memclass rettype
```

Required usage for function declaration and definition:
```
FUNC(void, @near) ExampleFunction (void);
```

9.3 Macros for pointers

9.3.1 P2VAR

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:	**COMPILER006**: 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:	--	

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;
```

9.3.2 P2CONST

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:	**COMPILER013**: 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:	--	

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;
```

9.3.3 CONSTP2VAR

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:	**COMPILER031**: 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:	--	

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;
```

9.3.4 CONSTP2CONST

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:	**COMPILER032**: 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:	--

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;
```

9.3.5 P2FUNC

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:	**COMPILER039**: The compiler abstraction shall define the P2FUNC macro for the type definition of pointers to functions.	
Caveats:	--	
Configuration:	--	

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);
```

9.4 Keywords for constants

9.4.1 CONST

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:	**COMPILER023**: The compiler abstraction shall define the CONST macro for the declaration and definition of constants.	
Caveats:	--	
Configuration:	--	

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;
```

9.5 Keywords for variables

9.5.1 VAR

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:	**COMPILER026**: The compiler abstraction shall define the VAR macro for the declaration and definition of variables.	
Caveats:	--	
Configuration:	--	

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;
```

10 Sequence diagrams

Not applicable.

11 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification, Chapter 11.1 describes fundamentals. We intend to leave Chapter 11.1 in the specification to guarantee comprehension.

Chapter 11.2 specifies the structure (containers) and the parameters of this module.

Chapter 11.3 specifies published information of this module.

11.1 How to read this chapter

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

- AUTOSAR Layered Software Architecture [3]
- AUTOSAR ECU Configuration Specification [4]. 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.

11.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or 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) shall be used in order to refer to a *specific configuration point in time*.

11.1.2 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.

Thus describe the possible configuration variants of this module. Each Variant must have a unique name which could be referenced to in later chapters. The maximum number of allowed variants is 3.

11.1.3 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

11.2 Containers and configuration parameters

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

11.2.1 Variants

Variant PC (**P**re **C**ompile): This is the only variant because all configuration parameters are pre-compile time parameters which influence the compilation process.
Each of the different memory classes (memclass) and pointer classes (ptrclass) is represented by a define.

SWS Item	**COMPILER044**
Container Name	<MSN>_MemoryAndPointerClasses
Description	This container contains the memory and pointer class parameters of a single module or of an application software component. For each module this container has to be provided. The number of different pointer and memory classes per module depends on the different types of variables, constants and pointers used by the module. It is allowed to extend the classes by module specific classes. The scope of all parameters is ECU because many parameters depend on the parameters of other modules. Examples for this are given in the Annex (starting on page 39). MSN means Module Abbreviation. For Application software components, the MSN is replaced by the component's name.
Configuration Parameters	

11.2.2 Module/Component Configuration (Memory and pointer classes)

Name	<MSN>_CODE		
Description	Configurable memory class for code.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		e.g. @near, _far
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_VAR_NOINIT		
Description	Configurable memory class for all global or static variables that are never initialized.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		e.g. @near, _far
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_VAR_POWER_ON_INIT		
Description	Configurable memory class for all global or static variables that are initialized only after power on reset.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		e.g. @near, _far
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_VAR_FAST		
Description	Configurable memory class 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		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		e.g. @near
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_VAR		
Description	Configurable memory class for all global or static variables that are initialized after every reset.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		e.g. @near
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_CONST		
Description	Configurable memory class for global or static constants.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_APPL_DATA		
Description	Configurable memory class for pointers to application data (expected to be in RAM or ROM) passed via API.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_APPL_CONST		
Description	Configurable memory class for pointers to application constants (expected to be certainly in ROM, for instance pointer of Init-function) passed via API.		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_APPL_CODE		
Description	Configurable memory class for pointers to application functions (e.g. call back function pointers).		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Name	<MSN>_CALLOUT_CODE		
Description	Configurable memory class for pointers to application functions (e.g. callout function pointers).		
Type	`#define`		
Unit	Compiler specific, refer to chapter 7		
Range	Compiler specific, refer to chapter 7		
Configuration Class	***Pre-compile***	x	Variant PC
	Link time	--	--
	Post Build	--	--
Scope	ECU		
Dependency	Memory Mapping		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
None	--	--

COMPILER042: The file Compiler.h is specific for each build scenario. Therefore there is no standardized configuration interface specified.

11.3 Published Information

[COMPILER001_PI] The standardized common published parameters as required by BSW00402 in the General Requirements on Basic Software Modules [2] shall be published within the header file of this module and need to be provided in the BSW Module Description. The according module abbreviation can be found in the List of Basic Software Modules [1].

Additional module-specific published parameters are listed below if applicable.

12 Annex

12.1 List of Compiler symbols

COMPILER012: The following table defines target compiler symbols according to COMPILER010. For each compiler supported by AUTOSAR a symbol has to be defined.

Platform	*Compiler*	*Compiler symbol*
S12X	Code Warrior	_CODEWARRIOR_C_S12X_
S12X	Cosmic	_COSMIC_C_S12X_
TC1796	Tasking	_TASKING_C_TC1796_
TC1766	Tasking	_TASKING_C_TC1766_
ST10	Tasking	_TASKING_C_ST10_
ST30	ARM Developer Suite	_ADS_C_ST30_
V850	Greenhills	_GREENHILLS_C_V850_
MPC5554	Diab Data	_DIABDATA_C_MPC5554_
TMS470	Texas Instruments	_TEXAS_INSTRUMENTS_C_TMS470_

12.2 Requirements on implementations using compiler abstraction

COMPILER040: Each AUTOSAR software module and application software component shall support the distinction of at least the following different memory classes and pointer classes.
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. The shortcut 'MSN' means 'module abbreviation of BSW module', e.g. 'EEP' or 'CAN'. It should be replaced by the component's name in case of a SoftwareComponent.

Memory type	*Syntax of memory class (memclass) and pointer class (ptrclass) macro parameter*	*Comments*	*Located in*
Code	<MSN>_CODE	To be used for code.	Compiler_Cfg.h
Constants	<MSN>_CONST	To be used for global or static constants	
Pointer	<MSN>_APPL_DATA	To be used for references on application data (expected to be in RAM or ROM) passed via API	
Pointer	<MSN>_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	<MSN>_APPL_CODE	To be used for references on application functions. (e.g. call back function pointers)	
Variables	<MSN>_CALLOUT_CODE	To be used for references on application functions. (e.g. callout function pointers)	
Variables	<MSN>_VAR_NOINIT	To be used for all global or static variables that are never initialized	

Memory type	*Syntax of memory class (memclass) and pointer class (ptrclass) macro parameter*	*Comments*	*Located in*
Variables	`<MSN>_VAR_POWER_ON_INIT`	To be used for all global or static variables that are initialized only after power on reset	
Variables	`<MSN>_VAR_FAST`	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	
Variables	`<MSN>_VAR`	To be used for global or static variables that are initialized after every reset.	
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

COMPILER041: 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:

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
);
```

12.3 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:

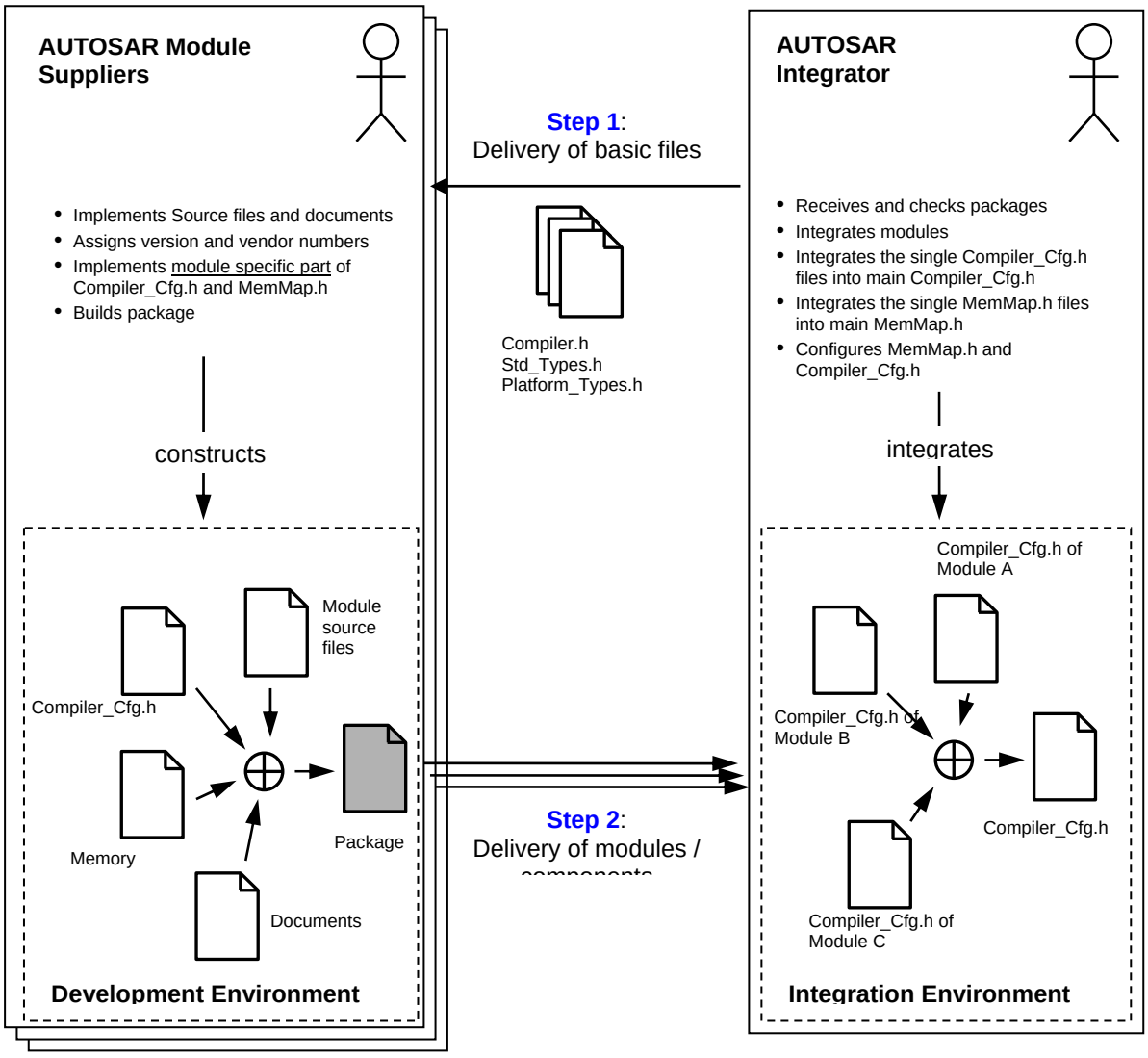

Figure 2: Proposal of integration-process

12.4 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: Metrowerks

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:

```
#include “Compiler.h”

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

What are the dependencies?
If 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 a MemMap_*.h.

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 32bit 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
```