| Document Title | Specification of C Implementation Rules |
|---|---|
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 190 |
| **Document Classification** | Auxiliary |

| | |
|---|---|
| **Document Version** | 1.0.5 |
| **Document Status** | Final |
| **Part of Release** | 4.0 |
| **Revision** | 1 |

**Document Change History**

| Date | Version | Changed by | Change Description |
|---|---|---|---|
| 03.12.2009 | 1.0.5 | AUTOSAR Administration | Legal disclaimer revised |
| 23.06.2008 | 1.0.4 | AUTOSAR Administration | Legal disclaimer revised |
| 31.10.2007 | 1.0.3 | AUTOSAR Administration | • Document meta information extended<br>• Small layout adaptations made |
| 24.01.2007 | 1.0.2 | AUTOSAR Administration | • “Advice for users” revised<br>• “Revision Information” added |
| 28.11.2006 | 1.0.1 | AUTOSAR Administration | Legal disclaimer revised |
| 09.05.2006 | 1.0.0 | AUTOSAR Administration | Initial Release |

## Disclaimer



## Advice for users:

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

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

# Table of Contents

# 1 Scope of this Document

This document contains implementation rules for the implementation language C. They are applicable for the development and maintenance of all AUTOSAR Basic Software modules that are written in C language.

Aims:

- Enhancing software quality by avoidance of risky language constructs
- Easier portability to other compilers or microcontroller platforms

This document applies to the whole software of an AUTOSAR ECU:

- AUTOSAR Basic Software
- RTE
- AUTOSAR Software Components

# 2 Acronyms and abbreviations

No acronyms and abbreviations that have a local scope are defined.

# 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

## 3.2 Related standards and norms

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

[4] ISO/IEC 9899:1990

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

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

[7] MISRA-C:2004

# 4 Constraints and assumptions

## 4.1 Limitations

No Limitations

## 4.2 Applicability to car domains

No restrictions.

## 4.3 Applicability to safety related environments

No restrictions.

# 5 Dependencies to other modules

No dependencies.

# 6 Requirements traceability

Document: AUTOSAR requirements on Basic Software, general

| ***Requirement*** | ***Satisfied by*** |
|---|---|
| [BSW00300] Module naming convention | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00301] Limit imported information | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00302] Limit exported information | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00304] AUTOSAR integer data types | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00305] Self-defined data types naming convention | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00306] Avoid direct use of compiler and platform specific keywords | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00307] Global variables naming convention | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00308] Definition of global data | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00309] Global data with read-only constraint | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00310] API naming convention | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00312] Shared code shall be reentrant | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00314] Separation of interrupt frames and service routines | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00318] Format of module version numbers | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00321] Enumeration of module version numbers | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00324] Do not use HIS I/O Library | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00325] Runtime of interrupt service routines | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00326] Transition from ISRs to OS tasks | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00327] Error values naming convention | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00328] Avoid duplication of code | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00329] Avoidance of generic interfaces | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00330] Usage of macros / inline functions instead of functions | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00331] Separation of error and status values | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00333] Documentation of callback function context | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00334] Provision of XML file | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00335] Status values naming convention | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00341] Microcontroller compatibility documentation | Not applicable<br>(as this is a rule document, not a module) |

| | |
|---|---|
| [BSW00342] Usage of source code and object code | Not applicable (as this is a rule document, not a module) |
| [BSW00343] Specification and configuration of time | Not applicable (as this is a rule document, not a module) |
| [BSW00341] Microcontroller compatibility documentation | Not applicable (as this is a rule document, not a module) |
| [BSW00346] Basic set of module files | Not applicable (as this is a rule document, not a module) |
| [BSW00347] Naming separation of different instances of BSW drivers | Not applicable (as this is a rule document, not a module) |
| [BSW00350] Development error detection keyword | Not applicable (as this is a rule document, not a module) |
| [BSW00353] Platform specific type header | Not applicable (as this is a rule document, not a module) |
| [BSW00355] Do not redefine AUTOSAR integer data types | Not applicable (as this is a rule document, not a module) |
| [BSW00350] Development error detection keyword | Not applicable (as this is a rule document, not a module) |
| [BSW00358] Return type of init() functions | Not applicable (as this is a rule document, not a module) |
| [BSW00359] Return type of callback functions | Not applicable (as this is a rule document, not a module) |
| [BSW00360] Parameters of callback functions | Not applicable (as this is a rule document, not a module) |
| [BSW00361] Compiler specific language extension header | Not applicable (as this is a rule document, not a module) |
| [BSW00370] Separation of callback interface from API | Not applicable (as this is a rule document, not a module) |
| [BSW00371] Do not pass function pointers via API | Not applicable (as this is a rule document, not a module) |
| [BSW00373] Main processing function naming convention | Not applicable (as this is a rule document, not a module) |
| [BSW00374] Module vendor identification | Not applicable (as this is a rule document, not a module) |
| [BSW00376] Return type and parameters of main processing functions | Not applicable (as this is a rule document, not a module) |
| [BSW00377] Module specific API return types | Not applicable (as this is a rule document, not a module) |
| [BSW00378] AUTOSAR boolean type | Not applicable (as this is a rule document, not a module) |
| [BSW00379] Module identification | Not applicable (as this is a rule document, not a module) |
| [BSW00401] Documentation of multiple instances of configuration parameters | Not applicable (as this is a rule document, not a module) |
| [BSW00408] Configuration parameter naming convention | Not applicable (as this is a rule document, not a module) |
| [BSW00410] Compiler switches shall have defined values | Not applicable (as this is a rule document, not a module) |
| [BSW00411] Get version info keyword | Not applicable (as this is a rule document, not a module) |
| [BSW00413] Accessing instances of BSW modules | Not applicable (as this is a rule document, not a module) |
| [BSW00414] Parameter of init function | Not applicable (as this is a rule document, not a module) |
| [BSW00415] User dependent include files | Not applicable (as this is a rule document, not a module) |
| [BSW005] No hard coded horizontal interfaces | Not applicable |

| | |
|---|---|
| within MCAL | (as this is a rule document, not a module) |
| [BSW006] Platform independency | Not applicable<br>(as this is a rule document, not a module) |
| [BSW007] HIS MISRA C | PROG_086 |
| [BSW009] Module User Documentation | Not applicable<br>(as this is a rule document, not a module) |
| [BSW010] Memory resource documentation | Not applicable<br>(as this is a rule document, not a module) |
| [BSW158] Separation of configuration from implementation | Not applicable<br>(as this is a rule document, not a module) |
| [BSW160] Human-readable configuration data | Not applicable<br>(as this is a rule document, not a module) |
| [BSW161] Microcontroller abstraction | Not applicable<br>(as this is a rule document, not a module) |
| [BSW162] ECU layout abstraction | Not applicable<br>(as this is a rule document, not a module) |
| [BSW164] Implementation of interrupt service routines | Not applicable<br>(as this is a rule document, not a module) |
| [BSW172] Compatibility and documentation of scheduling strategy | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00344] Reference to link-time configuration | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00404] Reference to post build time configuration | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00405] Reference to multiple configuration sets | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00345] Pre-compile-time configuration | Not applicable<br>(as this is a rule document, not a module) |
| [BSW159] Tool-based configuration | Not applicable<br>(this is a tool requirement) |
| [BSW167] Static configuration checking | Not applicable<br>(as this is a rule document, not a module) |
| [BSW171] Configurability of optional functionality | Not applicable<br>(as this is a rule document, not a module) |
| [BSW170] Data for reconfiguration of AUTOSAR SW-Components | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00380] Separate C-Files for configuration parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00419] Separate C-Files for pre-compile time configuration parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00381] Separate configuration header file for pre-compile time parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00412] Separate H-File for configuration parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00383] List dependencies of configuration files | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00384] List dependencies to other modules | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00387] Specify the configuration class of callback function | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00388] Introduce containers | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00389] Containers shall have names | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00390] Parameter content shall be unique within the module | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00391] Parameter shall have unique names | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00392] Parameters shall have a type | Not applicable |

| | |
|---|---|
| | (as this is a rule document, not a module) |
| [BSW00393] Parameters shall have a range | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00394] Specify the scope of the parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00395] List the required parameters (per parameter) | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00396] Configuration classes | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00397] Pre-compile-time parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00398] Link-time parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00399] Loadable Post-build time parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00400] Selectable Post-build time parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00402] Published information | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00375] Notification of wake-up reason | Not applicable<br>(as this is a rule document, not a module) |
| [BSW101] Initialization interface | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00416] Sequence of Initialization | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00406] Check module initialization | Not applicable<br>(as this is a rule document, not a module) |
| [BSW168] Diagnostic Interface of SW components | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00407] Function to read out published parameters | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00423] Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00424] BSW main processing function task allocation | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00425] Trigger conditions for schedulable objects | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00426] Exclusive areas in BSW modules | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00427] ISR description for BSW modules | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00428] Execution order dependencies of main processing functions | Not applicable<br>(not related to this specification) |
| [BSW00429] Restricted BSW OS functionality access | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00431] The BSW Scheduler module implements task bodies | Not applicable<br>(not related to this specification) |
| [BSW00432] Modules should have separate main processing functions for read/receive and write/transmit data path | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00433] Calling of main processing functions | Not applicable<br>(not related to this specification) |
| [BSW00434] The Schedule Module shall provide an API for exclusive areas | Not applicable<br>(not related to this specification) |

| | |
|---|---|
| [BSW00336] Shutdown interface | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00337] Classification of errors | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00338] Detection and Reporting of development errors | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00369] Do not return development error codes via API | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00339] Reporting of production relevant error status | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00348] Standard type header | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00357] Standard API return type | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00421] Reporting of production relevant error events | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00422] Debouncing of production relevant error status | Not applicable<br>(not related to this specification) |
| [BSW00420] Production relevant error event rate detection | Not applicable<br>(not related to this specification) |
| [BSW00417] Reporting of Error Events by Non-Basic Software | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00323] API parameter checking | Not applicable<br>(as this is a rule document, not a module) |
| [BSW004] Version check | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00409] Header files for production code error IDs | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00385] List possible error notifications | Not applicable<br>(as this is a rule document, not a module) |
| [BSW00386] Configuration for detecting an error | Not applicable<br>(as this is a rule document, not a module) |

# 7 Implementation rules

## 7.1 File

### 7.1.1 [PROG_003] File extensions

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | The interface file shall have the extension ".h" and the body file shall have the extension ".c". |
| ***Rationale:*** | Apply praxis |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

## 7.2 Code Presentation and Formatting

### 7.2.1 [PROG_000] English language

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 25.01.2005 |
| ***Importance:*** | Mandatory |
| ***Description:*** | The language within source files shall be English throughout. |
| ***Rationale:*** | -- |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.2.2 [PROG_008] Declarations per line

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Advisory |
| ***Description:*** | There should not be more than one declaration on a line. |
| ***Rationale:*** | Clear implementation |
| ***Example:*** | `uint8 blockIndex, lastBlock; /* violation */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.2.3 [PROG_087] Brackets in expressions

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Brackets shall always be used in complex expressions even if the C priority rules do not necessarily demand this for operators. This also applies to expressions evaluated by the preprocessor. |
| ***Rationale:*** | |
| ***Example:*** | `if ( (counter1 > 0) || (counter2 < 0) )`<br>`...`<br><br>`#if (SEC_SECURITY == C) || (SEC_SECURITY_CLASS == CCC)`<br>`...`<br>`#endif` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.2.4 [PROG_023] No space before and after '.'

| | |
|---|---|
| ***Initiator:*** | DC, BMW |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Do not insert a "blank" before (or after) the "." and the "->" operators. |
| ***Rationale:*** | This will harmonise the code style from different vendors, allowing easier reading and tool driven code checks |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.2.5 [PROG_024] No space between operators and operand

| | |
|---|---|
| ***Initiator:*** | DC, BMW |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Operators "++", "--", "&" (functionAddress) , "*" (FunctionRef)<br>shall be stuck to their operand. |
| ***Rationale:*** | Common notation for all Autosar source code makes it easier to review and automate for tool checks. |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.2.6 [PROG_025] No space after unary operator

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Unary operators "!" and "~" (operators that only have a right operand) shall be stuck to their operand. |
| ***Rationale:*** | Avoid confusion with a binary operator. |
| ***Example:*** | `if( ! computedBytes )  /* violation */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

## 7.3 Documentation and Commenting

### 7.3.1 [PROG_030] Commenting functions

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Function comments shall be positioned **in front of** the function header.<br>The function comments shall be above the functions in the C file. The function comments may also be in the H file. |
| ***Rationale:*** | Function comments in the H file are necessary if the module is to be delivered as object code. |
| ***Example:*** | `/*************************************************************`<br>`/* Function name:  MyFunc( uint8 X)`<br>`/* Description:    Short Description, including preconditions`<br>`/*   Parameter X : Description`<br>`/* Return value:   None`<br>`/* Remarks:        global variables used, side effects`<br>`/*************************************************************/`<br>`Void MyFunc( uint8 X)`<br>`{` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.3.2 [PROG_086] Commenting violations of MISRA rules

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Violations of MISRA rules shall be commented and reasoned at the corresponding code line. |
| ***Rationale:*** | |
| ***Example:*** | `/* MISRA RULE XX VIOLATION: This is the reason why the MISRA rule could not be followed in this special case */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.3.3 [PROG_090] Commenting violations of AUTOSAR C Implementation rules

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 26.01.2005 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Violations of AUTOSAR C programming guidelines shall be commented and reasoned at the corresponding code line. |
| ***Rationale:*** | |
| ***Example:*** | `/* AUTOSAR IMPL XX RULE VIOLATION: This is the reason why the rule could not be followed in this special case */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

## 7.4 Naming

### 7.4.1 [PROG_034] Double underscore

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Identifiers shall not contain the '_' character twice in succession. |
| ***Rationale:*** | Readability, allow system services |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.4.2 [PROG_038] Notation of macros

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Advisory |
| ***Description:*** | Constants defined as a macro shall be written in upper case. Digits and underscores are allowed but not at the start. |
| ***Rationale:*** | -- |
| ***Example:*** | `#define NR_OF_ELEMENTS   10` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

## 7.5 Includes

### 7.5.1 [PROG_042] Form of #include statements

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Header files which are a part of predefined program libraries shall be included using <>.<br><br>Header files which are a part of the source code generated in the software project shall be included with "". |
| ***Rationale:*** | Separation of namespace |
| ***Example:*** | `#include <string.h>;`<br><br>`#include "Eep.h";` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.5.2 [PROG_044] Protection against multiple inclusion

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Each header file shall protect itself against multiple inclusion. |
| ***Rationale:*** | Avoid multiple re-definitions. |
| ***Example:*** | `#ifndef FILENAME_H`<br>`#define FILENAME_H`<br>`.....`<br>`#endif /* FILENAME_H */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.5.3 [PROG_048] Inclusion of own header file

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Each module shall include its own header file. |
| ***Rationale:*** | This is the only way of allowing the compiler to check for consistency between declaration and definition of global variables and functions. |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.5.4 [PROG_050] Body inclusion

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | A ".c" file shall not be included in another file: it shall be compiled and provided as an object module. |
| ***Rationale:*** | Reduce include effort |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

## 7.6 Resource

### 7.6.1 [PROG_055] Declarations of global functions

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | A declaration with storage-class specifier "extern" shall exist for each global function in the header file of the module. |
| ***Rationale:*** | Allow access to global functions by including the header file of the defining module. |
| ***Example:*** | `extern Std_ReturnType Eep_Erase`<br>`(`<br>`  Eep_AddressType  EepromAddress,`<br>`  Eep_LengthType  Length`<br>`);` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.6.2 [PROG_063] Declarations of global variables

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | External declarations of global variables shall be done in header files and never in .c files. |
| ***Rationale:*** | Provide access to global variables by including the header file of the module that defines the global variables. |
| ***Example:*** | `extern Com_TypeX Com_Global_Var;` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.6.3 [PROG_061] No function definition within header file

| | |
|---|---|
| ***Initiator:*** | DC, BMW |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Functions (other than macros) shall not be defined within in a ".h" file.<br><br>Exception:<br>Definitions of inline functions in the header file are permitted. |
| ***Rationale:*** | Reduced include effort |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.6.4 [PROG_057] No variable definition within header file

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Variables shall not be defined within in a ".h" file. They shall be defined within the module's C file. |
| ***Rationale:*** | Prevent multiple variables, i.e. linker problem |
| ***Example:*** | -- |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.6.5 [PROG_062] Declaration of function parameters

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Declarations of functions shall always be stated with detailed parameter list, i.e. the type and a practical designation of the relevant parameters.<br><br>Designator names in C and H file shall be identical. |
| ***Rationale:*** | Readability and correct use of API. |
| ***Example:*** | `extern Std_ReturnType Eep_Erase`<br>`(`<br>`  Eep_AddressType  EepromAddress,`<br>`  Eep_LengthType  Length`<br>`);` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.6.6 [PROG_056] Declaration and definition of local functions

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Declaration and definition of local functions shall have the storage-class specifier “static”.<br><br>Local function means function with internal linkage (only visible inside the module). |
| ***Rationale:*** | Limit visibility of local functions. |
| ***Example:*** | `static void MyLocalFunction( void);` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.6.7 [PROG_058] Explicit definition of types

| | |
|---|---|
| ***Initiator:*** | BMW |
| ***Date:*** | 13.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Each self-defined type has to have an explicit type declaration even if there is only one variable of this type. |
| ***Rationale:*** | -- |
| ***Example:*** | `typedef struct`<br>`{`<br>`  uint16 Position;`<br>`  uint8  Direction;`<br>`} MotorType;`<br><br>`static MotorType MotorData;` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

## 7.7 Complexity

### 7.7.1 [PROG_071] Multiple Assignment

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Mandatory |
| ***Description:*** | Multiple assignments shall not be done. |
| ***Rationale:*** | -- |
| ***Example:*** | `x = y = z;  /* violation */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

### 7.7.2 [PROG_072] Use of '++' and '- -'

| | |
|---|---|
| ***Initiator:*** | DC |
| ***Date:*** | 15.10.2004 |
| ***Importance:*** | Advisory |
| ***Description:*** | The use of '++' and '- -' should be limited to simple cases. They shall not be used in statements where other operators occur. The prefix use is always forbidden. |
| ***Rationale:*** | This rule only belongs to '++' and '- -'. Statements like '+=' are excluded. |
| ***Example:*** | `x -= y++;  /* violation */` |
| ***Conflicts:*** | -- |
| ***Dependencies:*** | -- |

# 8 Configuration Implementation Guidelines

## 8.1 Introduction

This chapter is a set of guidelines for the implementation of the various configuration options for the BSW in AUTOSAR. Therefore this chapter is of direct relevance for anyone implementing BSW modules.

This chapter is in 3 main sections, as follows. Section 8.2 discusses what aspects of the code and ECU we are trying to represent and some underlying assumptions. Section 8.3 discusses the role of variants, as described in the BSW SWS documents. Section 8.4 discusses how configurations are represented in source and object code. This is the section of most direct relevance to implementers.

This chapter is not a requirements specification. Neither is it normative. It is a set of guidelines. In places the AUTOSAR C rules have been used, but this is not universal. Also, no attempt has been made to use standardized file names. It must be understood the not adhering to these guidelines may result in BSW that does not integrate with other AUTOSAR BSW.

## 8.2 Scope of Guidelines

The purpose of this document is to guide BSW implementers on the correct choice of implementation strategy for the BSW modules. Therefore these guidelines first consider the different configuration classes (pre-compile, post-link, etc.) and their relation to the variants. Their representation in source code and memory (code memory, non-volatile data memory and RAM) is then considered along with any tooling issues that are necessary to get the appropriate code and data into memory.

Code examples are used to illustrate the various concepts so that, by the end of this chapter, it should be possible to work out the code, memory and tooling issues required to support the implementation of a BSW module.

### 8.2.1 Assumptions

This section defines some of the assumptions that we made when writing these guidelines.

> Assumption 1: Any configuration that takes place post compilation will not be able to change any code. Only data can be changed after compilation.
>
> Assumption 2: Data will be packed into memory in an efficient manner. This implies that, should an area of configuration data change size, the start addresses of any subsequent data might also change. Therefore pointers set at run-time to point to configuration data shall be used. This may mean that code overheads increase under certain circumstances.

Assumption 3: We assume that only the BSW below the RTE will be configurable using the methods described in this chapter. This means that the RTE may not be configurable using these methods. (i.e. the RTE will be able to change its configuration but we make no statement about how it may do so in this chapter.) This implies that the interface between the RTE and COM cannot be changed by reconfiguration of COM alone. (i.e. the signals and callbacks must remain the same.)

Further assumptions refer to specific configuration variants and are documented in the relevant sections.

### 8.2.2 Memory types

It is assumed that an ECU contains RAM and a non-volatile code and data memory. The RAM will be referred to as RAM. The non-volatile code and data memory will be referred to as FLASH.

We assume that code cannot be changed once it has been compiled.

We assume that data can be changed once the code has been compiled, but only provided that the data has been structured to allow the changes to take place.

### 8.2.3 Stages in tooling

We assume that the normal (for a C compiler) tool stages exist.

A code generator is run that reads in the XML for a particular module and produced on its output code, data, or both, that complete the implementation of that module.

A pre-processor exists so that macro substitutions and code removal can take place before compilation.

A compiler exists that produces object code. We do not assume the existence of an intermediate assembly language representation.

A linker that combines one or more object modules, resolves references, and allows code and data segments to be placed in memory.

As necessary we will also describe, where necessary, additional tools to support configuration activities.

## 8.3 Role of Variants

There are four Configuration Classes defined in the Software Layered Architecture.

Pre compile time. All code and data are determined before the compiler is run. The code and data are then placed into the ECU's FLASH at the same time.

Link time. Code is compiled with data added later by the linker. The code and data are then placed into the ECU's FLASH at the same time.

Post build time. Code is compiled with pointers into blank areas of FLASH. The compiled code is then placed into FLASH. The data is placed into FLASH at a different stage in the manufacturing process than the code. The data to be used at run-time can either be fixed (this is called post build loadable) or selected at run-time (this is called post build selectable). Note that these two classes are mutually exclusive. You cannot have a mixture of post build loadable and post build selectable in a particular variant in a particular module.

The variants allow combinations of the configuration classes to be defined to allow flexibility in the implementation. The variant concept is now illustrated with some examples. Note that these are **examples**. Individual WPs may make different decisions about what the different variants mean.

**Example variant 1: Most efficient possible implementation - everything is pre compile time.**
In this variant all parameters are determined pre compile time. There is no possibility to change any parameters once compilation has taken place (other than by recompiling). Therefore this variant has the capability to be the most efficient but is also the least flexible. A good example of the pre compile parameter is DevelopmentErrorChecking. This is typically implemented with the pre-processor to enable or disable code generation for specific areas of the module's source. However, if you change your mind then you need to recompile.

**Example variant 2 - Adding flexibility with link time parameters.**
In this variant flexibility is added by choosing some parameters that will be determined at link time. Therefore, typically, a mixture of pre compile and link time parameters will exist in this variant. The pre compile parameters are set before compilation, as in variant 1, and can only be changed by recompilation. Again the DevelopmentErrorChecking parameter is a good example here. Therefore, the module would typically be delivered as source code. Other parameters can be set at link time without the need to recompile the module. For example, the Baud rate in a CAN controller might be determined after compilation by making this a link time parameter.

In a particular variant a parameter can only by in one configuration class. It is not possible to have a parameter in more than one configuration class in a particular variant. However, a parameter may change from one class to another between two variants.

## 8.4 Structure at compile-time and run-time

This section describes how each configuration class works and how they might be combined into different variants. There are some initial assumptions that are first explained, and then each configuration class's implementation is described along with how these are combined into a variant.

### 8.4.1 Common factors

All of these examples concentrate upon the following aspects: what source code files are required, what header files are required, what the stages in the tooling are, how FLASH (for both code and data) is allocated, how RAM is allocated and how IO devices are accessed and initialized.

The references between code and data are shown diagrammatically in Figure 1. In this figure, an arrow from box A to box B means that A must be able to contain some sort of reference to B so that A can either read or write B, or so that code can deduce the something in B from the contents of A.

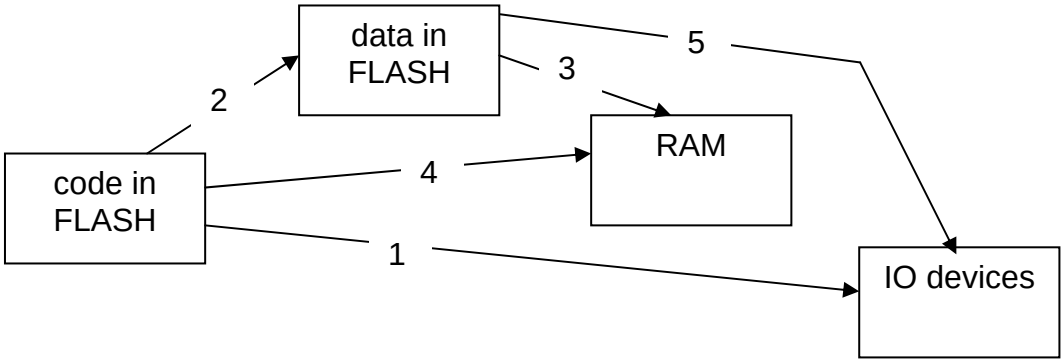


**Figure 1. Different code/data areas and accesses.**

The purposes of the arrows are now explained in more detail. The numbers below refer to those on the arrows.

1) Code needs to access IO devices for reading and writing. The address of the IO device can either appear as a constant in the code, or can be read from constant data, or can be calculated at run-time.

2) Code needs to access constant data for configuration items. For example, the constant data might contain the period of a frame on the network. The period will never change as it's in constant data.

3) The constant data might also contain pointers into RAM. For example, some modules need RAM buffers to store data. One way to access these is to store the address of the buffer in constant data.

4) Another way to access RAM data is to store the address in the code.

5) Similarly, addresses of IO devices might be contained in FLASH. This allows the specific IO device to be used to be changed after the code has been built.

In the following discussion possible implementations of these data pathways will be discussed. Not all pathways can be implemented in all configuration classes.

A strong distinction between a datum's definition and its declaration also needs to be made. The usual C distinction is used where the *declaration* states the type and *definition* allocates the storage.

There is also a distinction between a module and a compilation unit. In this chapter a module is a complete BSW module. A compilation unit is a single C source file that is compiled resulting in an object file. The implementation of a module will typically consist of many compilation units that are linked together.

In the following example a strong distinction is made between RAM and FLASH. Data that needs to be placed in FLASH uses the const qualifier. If the const qualifier is omitted then the data resides in RAM.

We now consider how the different configuration classes are implemented. The following sections contain some very detailed code examples and will need careful study.

### 8.4.2 Pre compile time implementation

This is the simplest option and can result in the most efficient implementation. This is because the code generator can produce code and data that is optimised for the configuration being generated.

The entire module has to be delivered as source code. In this example there are four source files:

**xx.c** This is part of the implementation of the module and, therefore, part of the delivery of the module and would not normally be changed by the user of the module.

**xx_structs.h** These are data structure declarations that the module's implementation uses. This is part of the delivery of the module and, therefore, would not normally be changed by the user of the module.

**xx_cfg.h** This file is generated automatically by the code generator from the AUTOSAR XML. It just contains macro definitions. The only changes to this file can be made by altering the XML and regenerating the file.

**xx_cfg.c** This file is generated automatically by the code generator from the AUTOSAR XML. It contains definitions of data used by xx.c.

Note that page breaks have been inserted into the following examples so that a single file does not cross a page boundary in order to improve readability.

In the following source codes the pre compile switches are implemented exemplary by using "`#ifdef`". Note that defined values should be used according to the general requirement BSW00410: `#if (MY_SWITCH == STD_ON)`.

Note that the following source codes use exemplary filenames which do not completely follow the general requirement BSW00346, which requests the names "`xx_Lcfg.c`" for link time configuration parameters and "`xx_PBcfg.c`" for post build configuration parameters.

**xx.c**

```
#include "xx_structs.h"
#include "xx_cfg.h"

void f(void) {
      unsigned int count;

#ifdef DEVELOPMENT_ERROR_CHECKING
      if(!module_initialised) {
            /* do something with the DET */
            return;
      }
#endif

      /* set CAN Baud rate */
      *BAUD_RATE_REGISTER = BAUD_RATE;

      /* initialise the buffers in RAM from data in ROM*/
      for(count=0; count<NBUFFS; count++) {
#ifdef CACHING_IS_BETTER
            /* this code caches the pointer */
            struct buffer *tmp = &my_configuration[count];
            xx_memcopy(tmp->size, tmp->init_data, tmp->ram_data);
#else
            /* Caching is not better */
            xx_memcopy(my_configuration[count]->size,
                              my_configuration[count]->init_data,
                              my_configuration[count]->ram_data);
#endif /* CACHING_IS_BETTER */
      }
}
```

### xx_structs.h

```
/* Data structures that are part of the implementation
 * and should not be changed.
 */
struct buffer {
      /* size of data in this buffer */
      unsigned int size;
      /* pointer to initial value for data in buffer */
      const unsigned char *init_data;
      /* pointer to ram for buffer */
      unsigned char *ram_data;
};

/* flag to state whether or not this module has been initialised.*/
extern unsigned char module_initialised;

extern const struct buffer my_configuration[];

void xx_memcopy(unsigned int size,
                        const unsigned char *src,
                        unsigned char *dest);
```

### xx_cfg.h

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

/* These defines may be automatically generated depending
 * upon the module implementation
 */
#define BAUD_RATE (0xf0)
#define BAUD_RATE_REGISTER ((unsigned char *)(0xfe00))

/* DET is turned on in the XML. This define would be omitted
 * if DET is not required
 */
#define DEVELOPMENT_ERROR_CHECKING

/* number of entries in my_configuration */
#define NBUFFS ((unsigned int)2)

/* Pointer caching is better on this target */
#define CACHING_IS_BETTER
```

**xx_cfg.c**

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

#include "xx_structs.h"

/* RAM buffers for different variables */
unsigned char buffer_bob[4];
unsigned char buffer_bill[3];

/* ROM initial values for buffers */
const unsigned char initial_bob[] = {0x0, 0x23, 0x12, 0xe4};
const unsigned char initial_bill[] = {0xdf, 0x3d, 0x12};

/* ROM generated configuration information for module */
const struct buffer my_configuration[] = {
      {4, initial_bob, buffer_bob},       /* buffer for bob */
      {3, initial_bill, buffer_bill},     /* buffer for bill */
};
```

The build process for these files would be as follows:

1. The code generator for this module would be run to generate xx_cfg.h and xx_cfg.c.
2. xx_cfg.c would be compiled because it changed in step 1.
3. xx.c would be compiled because xx_cfg.h (upon which it depends) changed in step 1.
4. All modules would be linked because they changed in steps 2 and 3.

There are some interesting points in this implementation that will now be pointed out.

In xx_cfg.h the number of buffers is represented as a macro. This is more efficient than using a variable (as depicted in section 8.4.3) but also means that xx.c needs to be recompiled if the number of buffers changes.

In xx.c five macros are used to govern where an IO device is, to change some code generation options and set the size of a buffer. If the IO device moves, or the Baud rate changes, the file must be recompiled. Also, if you are experimenting with code efficiency it is necessary to compile with CACHING_IS_BETTER on and off and compare the results.

### 8.4.3 Link time implementation

In a link time implementation the major restriction is that xx.c will be delivered as object code and is therefore immutable by the user. This means that some of the macros in the pre compile time example need to be turned into variables. There were five macros in the pre compile time version. Which of those are turned into variables is specified by the variant information and is worked out by the WG that writes the SWS for the module. i.e. The parameter is moved from the pre compile configuration class to link time in its declaration in the SWS.

Because the module is delivered as object code, any data that changes with the configuration needs to be expressed as a variable. The module's source imports the variable by declaring it. The value is set in another C file that declares and initializes it.

Three files are needed for this: two .c files and a header file that ensures consistency between the .c files. Brief examples are shown below.

**link_time.h**
```
extern const int fred;
```

**user.c**
```
#include "link_time.h"
int get_fred(void) {
        return fred;
}
```

**init.c**
```
#include "link_time.h"
const int fred = 0x1234;
```

The file user.c uses fred but does not know in advance what its value is. The file is compiled by the supplier of the module and, therefore, cannot by changed by the user of the module. The file link_time.h is also supplied as part of the module, because it is necessary to compile a file in the user's domain, and cannot be changed by the user.

The file init.c is generated and compiled by the user. When user.o and init.o are linked the reference by user.c to fred is resolved so that the function get_fred() can work correctly.

From a tools point-of-view this works as follows:

1. Module supplier uses the code generator to generate link_time.h
2. Module supplier compiles user.c
3. Module supplier provides user.o and link_time.h to user.

There can be a significant gap between steps 3 and 4 because they may take place at different times and in different divisions or companies.

4. User uses the code generator to generate init.c
5. User compiles init.c
6. User links user.o and init.o

The next section shown how pre compile and link time can be united in the same variant.

### 8.4.4 Combining pre compile and link time

This combination is illustrated using the example from section 8.4.2. Small changes to this example are made that move some of the compile time constants into link time values. The result is less efficient but more flexible.

The file are very similar in content to those shown in section 8.4.2. The main differences are now described.

**xx.c** This is the implementation of the module. This is part of the delivery of the module and is built by the supplier of the module. All data that is configurable by the user has been removed and placed in other files. However, the macros for the DET and pointer caching are still present and need to be dealt with.

The DET macro implies that this module will be supplied in a number of flavours. This is because there are typically several macros like this per module. So the user and supplier need to agree in advance which flavours will be supplied. All the source files for that module will be compiled by the supplier with these macros set to the same value, and then a library file created. So there will be as many library files as flavours.

The macro NBUFFS has been transformed into the variable nbuffs, and the macro BAUD_RATE has been transformed into the variable baud_rate. This is typically less efficient than using macros. However, these parameters can now be set *after* xx.c has been compiled. This was not the case in the pre compile implementation.

**xx_structs.h** These are data structure declarations that the module's implementation uses. In this version the Baud rate and number of buffers have been added. This is part of the delivery of the module and, therefore, would not normally be changed by the user of the module.

**xx_cfg.h** This file is generated automatically by the code generator from the AUTOSAR XML. It just contains macro definitions. The only changes to this file can be made (in ideal situations) by the supplier. This header file has to agree with the contents of xx.c.

**xx_cfg.c** This file is generated automatically by the code generator from the AUTOSAR XML. It contains definitions of data used by xx.c. In this version I have added the Baud rate and number of buffers.

**xx.c**

```
#include "xx_structs.h"
#include "xx_cfg.h"

void f(void) {
      unsigned int count;

#ifdef DEVELOPMENT_ERROR_CHECKING
      if(!module_initialised) {
            /* do something with the DET */
      }
#endif

      /* set CAN Baud rate, this time with an
       * external value.
       */
      *BAUD_RATE_REGISTER = baud_rate;

      /* Initialise the buffers. The number of buffers to initialise
       * is set externally.
       */
      for(count=0; count<nbuffs; count++) {
#ifdef CACHING_IS_BETTER
            /* this code caches the pointer */
            struct buffer *tmp = &my_configuration[count];
            xx_memcopy(tmp->size, tmp->init_data, tmp->ram_data);
#else
            /* Caching is not better */
            xx_memcopy(my_configuration[count]->size,
                              my_configuration[count]->init_data,
                              my_configuration[count]->ram_data);
#endif /* CACHING_IS_BETTER */
      }
}
```

**xx_structs.h**

```
/* ROM initialiser for Baud rate */
extern const unsigned char baud_rate;

/* ROM initialiser for number of entries in my_configuration */
extern const unsigned int nbuffs;

extern const struct buffer my_configuration[];

void xx_memcopy(unsigned int size,
                        const unsigned char *src,
                        unsigned char *dest);
```

**xx_cfg.h**

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

/* These defines may be automatically generated depending
 * upon the module implementation
 */
#define BAUD_RATE_REGISTER ((unsigned char *)(0xfe00))

/* DET is turned on in the XML. This define would be omitted
```

```
 * if DET is not required */
#define DEVELOPMENT_ERROR_CHECKING

/* Pointer caching is better on this target */
#define CACHING_IS_BETTER
```

**xx_cfg.c**

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

#include "xx_structs.h"

/* RAM buffers for different variables */
unsigned char buffer_bob[4];
unsigned char buffer_bill[3];

/* ROM initial values for buffers */
const unsigned char initial_bob[] = {0x0, 0x23, 0x12, 0xe4};
const unsigned char initial_bill[] = {0xdf, 0x3d, 0x12};

/* ROM generated configuration information for module */
const struct buffer my_configuration[] = {
      {4, initial_bob, buffer_bob},       /* buffer for bob */
      {3, initial_bill, buffer_bill},     /* buffer for bill */
};

/* ROM initialiser for Baud rate */
const unsigned char baud_rate = (0xf0);

/* ROM initialiser for number of entries in my_configuration */
const unsigned int nbuffs = (2);
```

The build process for this implementation is as follows:

1. Module supplier builds XML to configure pre compile parameters
2. Module supplier uses code generator to generate xx_cfg.h
3. Module supplier compiles xx.c and puts it in a library.

Typically there will be more than one library, each library will correspond to a different collection of build options in pre processor macros. Putting header files (necessary for building the files in later steps) in a library is not easily achieved. So the header files will need to be written so that the pre processor includes and excludes lines of code correctly for different build flavours.

4. Module user takes delivery of a collection of libraries.
5. Module user configures XML to generate link time data
6. Module user runs code generator to generate xx_cfg.c.
7. Module user compiles xx_cfg.c
8. Module user links xx_cfg.o with appropriate library to produce the application.

There are some interesting points in this example. The first is that there is not a huge difference in source code between this and the pre compile case. This is because the main bulk of the configuration data, which will be contained in tables and other structures is usually in a separate compilation unit.

The main difference is in the efficiency of the function f(). The baud rate and size of data tables are obtained from configuration memory in this case resulting in a slightly less efficient but more versatile implementation.

Another interesting point is the optimization macro, CACHING_IS_BETTER. Because the module is delivered as object code it is now harder to optimize the value of this macro. This can lead to greater inefficiency.

However, we now have a distinction between configuration items that that supplier of the module can set, and those that the user of the module can set.

RAM is allocated by the linker in this scheme.

### 8.4.5 Post build implementation

The purpose behind post build is to allow one or more configurations to be loaded into FLASH at a late point in the manufacture of the ECU. So the ECU might be built and programmed by a vendor but delivered to the OEM with some configuration data missing. The OEM then develops this configuration data and writes it into the ECU's FLASH before deployment in the vehicle.

The ECU vendor's software needs to know where to find the configuration data. This is typically achieved by writing the data to fixed point in FLASH. So some blocks of flash are set aside just for configuration information, and the module built so that it knows about the place in FLASH where the configuration information starts.

In AUTOSAR it is likely that more than one module will be post build configured. One possible way to store the configuration information is to allocate a section of FLASH to each BSW module with headroom should that module grow. However, the number of modules involved makes this prohibitive in practice due to amount of memory that would be wasted. i.e. This idea violates assumption 2 in section 8.2.1.

So we need to pack individual configuration data in an efficient manner. So first we'll examine how to represent the configuration of a single module, then how to pack many modules efficiently, and then how to represent more than one configuration for each module.

#### 8.4.5.1 Single module configuration

In order to represent a single module the scalar data for that module, and the starting points of all complex structures, are collected into a structure. For example, a module that has:

unsigned int lower;
unsigned int upper;

as configuration data would be transformed into:

```
struct xx_configuration {
        unsigned int lower;
        unsigned int upper;
};
```

An instance of this structure can be statically initialized in its own C source file, compiled and then the linker used to place it at the correct place in memory. For example, it might be placed at 0x8000.

The module's source would need to be changed so that access to its configuration takes place via a pointer to the configuration structure. For example:

```
struct xx_configuration *config = (struct xx_configuration *)(0x8000);
my_upper = config->upper;
```

This scheme works for a small number of modules. But rapidly becomes very inefficient in memory usage as the number of modules increases. So the modification in the next subsection is used.

#### 8.4.5.2 Packing the configuration for many modules into memory

When the configuration data for many modules needs to be packed into memory the instances of the xx_configuration (and yy_configuration, and zz... for each module) are placed sequentially in memory with an index at the front.

The index looks like this:

```
struct module index_tag {
        struct xx_configuration *xx_conf;
        struct yy_configuration *yy_conf;
        struct z_configuration *zz_conf;
        /* etc for each module */
} index = {
        &the_real_xx_configuration;
        &the_real_yy_configuration;
        &the_real_zz_configuration;
        /* etc for each module */
};
```

The index structure is placed at the start of FLASH at a known address using the linker.

The tooling for this method of configuration, as seen from the OEM's view point, is as follows:

1. Configure XML for system.
2. For each module
3. Use the code generator to generate the configuration data for that module
4. Compile the configuration data
5. End of “For each”
6. Use the code generator to generate the index table
7. Compile the index table.
8. Link the objects for the index table and module configurations ensuring that the index is first in memory and at the right place.
9. Blow the resulting data into FLASH.

During initialization the ECU's software (the EcuM and ComM) reads the index to find out where each module's initialization data is, and calls the module's xxx_init() function with a pointer to that data.

> Assumption 5: The EcuM and ComM will need to work out the correct pointers for the xx_init() functions and call these functions.

The implementation of each module must therefore change so that, for post build parameters, a pointer to the module's configuration is used.

#### 8.4.5.3 Selecting from more than one configuration for each module

Once the module's implementation has been changed to accept a pointer to post build configuration data, that pointer can be from a variety of sources. There need not be a single index for the entire system. A system that needs to be deployed in more than one place in the vehicle can be set up so that that there is a separate configuration for each position, and the configuration is selected at run-time.

The next section shows example code that supports pre compile, link time and post build configurations simultaneously.

#### 8.4.5.4 Combining pre compile, link time and post build

The example from section 8.4.4 has been adapted to move some of the configuration into the post build category. An extra file was added to contain the post build information. The files are as follows:

> **xx.c** This is the implementation of the module. In this example I've changed the function to be the xx_init() function so that passing a pointer to the configuration is more easily demonstrated. The pointer to the configuration is

cached by the xx_init() function so that other compilation units in this BSW module can use the pointer. This uses more RAM but is also faster than rediscovering the pointer each time it is needed.

There is still a #ifdef for the DET. So this module will need to be shipped in more than one flavour, as in the previous example implementations.

**xx_structs.h** These are data structure declarations that the module's implementation uses. This is part of the delivery of the module and, therefore, would not normally be changed by the user of the module. In this example this file has been extended to include some variables that need to be shared by all compilation units in the module and also to include the structure that is the root of the configuration data for the module.

**xx_cfg.h** This file is generated automatically by the code generator from the AUTOSAR XML. It just contains macro definitions needed to compile xx.c.

**xx_cfg.c** This file is generated automatically by the code generator from the AUTOSAR XML. It contains definitions of data used by xx.c at link time.

**xx_pb.c** This file contains the post build configuration data. The root of the configuration is in the data called real_xx_configuration. A pointer to this should be passed into the xx_init() function. From this structure all the post build configuration is found.

This file is compiled and linked separately from the other .c files. The linker is used to place it in FLASH.

**xx.c**

```
#include "xx_structs.h"
#include "xx_cfg.h"

/* Allocate memory for this pointer cache.
 */
struct xx_configuration *xx_config_cache;

/* Allocate memory for initialised flag
 */
unsigned char module_initialised;

void xx_init(struct xx_configuration *config) {
      unsigned int count;

#ifdef DEVELOPMENT_ERROR_CHECKING
      /* Check for an attempt to initialise a module that
       * is already in use.
       */
      if(0 != module_initialised) {
            /* do something with DET */
            return;
      }
#endif /* DEVELOPMENT_ERROR_CHECKING */

      /* cache the pointer to the configuration. Any of the other
       * compilation units in the BSW may need this pointer.
       */
      xx_config_cache = config;

      /* access to a link time allocated variable
       */
      module_initialised = 1;       /* i.e. TRUE... */


      /* set CAN Baud rate with a link time value.
       */
      *BAUD_RATE_REGISTER = baud_rate;

      /* Initialise the buffers. The number of buffers to initialise
       * and the buffer positions, are set via a post build structure.
       */
      for(count=0; count<xx_config_cache->nbuffs; count++) {
            /* this code caches the pointer */
            struct buffer *tmp =
                  &xx_config_cache->my_configuration_ref[count];
            xx_memcopy(tmp->size, tmp->init_data, tmp->ram_data);
      }
}
```

**xx_structs.h**

```
/* Data structures that are part of the implementation
 * and should not be changed.
 */

/* flag to state whether or not this module has been initialised.*/
/* this is in RAM that is allocated by the linker at the link time
 * configuration stage
 */
extern unsigned char module_initialised;

/* ROM initialiser for Baud rate. Just for this example the
 * Baud rate is still link time configurable. It is not
 * configurable at post build time.
 */
extern const unsigned char baud_rate;

/* Pointer cache. Allocated memory by the linker when
 * xx.c is linked. Appears in this file so that other
 * compilation units in this module can use it.
 */
extern struct xx_configuration *xx_config_cache;

struct buffer {
      /* size of data in this buffer */
      unsigned int size;
      /* pointer to initial value for data in buffer */
      unsigned char *init_data;
      /* pointer to ram for buffer */
      unsigned char *ram_data;
};


/* This structure contains the root of all the post build time
 * configuration data for this module.
 */
struct xx_configuration {
      /* initialiser for number of entries in my_configuration */
      unsigned int nbuffs;

      /* pointer to configuration data
       */
      struct buffer *my_configuration_ref;
};

void xx_memcopy(unsigned int size,
                        const unsigned char *src,
                        unsigned char *dest);
```

**xx_cfg.h**

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

/* These defines may be automatically generated depending
 * upon the module implementation
 */
#define BAUD_RATE_REGISTER ((unsigned char *)(0xfe00))

/* DET is turned on in the XML. This define would be omitted
 * if DET is not required */
#define DEVELOPMENT_ERROR_CHECKING
```

**xx_cfg.c**

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

/* This file contains data allocated at link time
 */

#include "xx_structs.h"


/* ROM initialiser for Baud rate.
 * This is the only remaining value allocated at
 * the module's link time in this example.
 */
const unsigned char baud_rate = (0xf0);
```

**xx_pb.c**

```
/* AUTOMATICALLY GENERATED FILE - DO NOT EDIT */

#include "xx_structs.h"

/* This file contains parameters generated as part of the post build
 * configuration.
 */

/* RAM buffers for different variables */
unsigned char buffer_bob[4];
unsigned char buffer_bill[3];

/* ROM initial values for buffers */
const unsigned char initial_bob[] = {0x0, 0x23, 0x12, 0xe4};
const unsigned char initial_bill[] = {0xdf, 0x3d, 0x12};

/* ROM generated configuration information for module */
const struct buffer my_configuration[] = {
      {4, initial_bob, buffer_bob}, /* buffer for bob */
      {3, initial_bill, buffer_bill},     /* buffer for bill */
};

/* ROM generated root configuration for this module
 */
const struct xx_configuration real_xx_configuration = {
      2,                            /* number of buffers */
      my_configuration  /* buffer structures */
};
```

From a process point-of-view there are three stages to the configuration of the ECU:

1. Build the module using pre compile configuration. Sets the address of the baud rate register and other compile-time options. Typically performed by the module's supplier.
2. Build the ECU software using link-time options. Sets the baud rate in this example. Typically performed by the tier 1 supplier.
3. Final configuration for the vehicle using post build. Sets the buffers, their initial values, etc. in this example. Typically performed by the OEM.

However, this still omits the index structure referred to in section 8.4.5.2. This omission is now addressed.

At run-time a BSW module's xx_init() function is passed a pointer that points to the module's post build configuration data. How is this pointer determined? Most BSW modules have their xx_init() functions called either from the EcuM or (in the case of the communications stack) the ComM. The EcuM calls the ComM. So the EcuM must be able to work out what the pointers are for each module.

In order to do this the EcuM has an index structure that contains the pointers for all BSW modules that are to be post build configured. The overall structure is shown diagrammatically in Figure 2.

**Figure 2. Diagrammatic representation of post build loadable.**

In order to implement this we are missing the EcuM code and the index. Example files for these are now presented. The files are as follows:

**ecum_init.c** This file contains a function called by the EcuM that initializes the BSW modules. This file is automatically generated from the XML's list of modules that are post build configurable and compiled by the tier 1.

**ecum.h** This file contains the definition of the index data structure. This file is automatically generated from the XML's list of modules that are post build configurable by the tier 1.

**pb_ecum.c** This file contains the initialization data for the index data structure. This file is automatically generated from the XML's list of modules that are post build configurable by the tier 1. The linker's control file **must ensure** that the index is at a known address in FLASH. Therefore the linker's control file will probably be generated from the XML too by the tier 1 or by the OEM. The XML and code generation must ensure that they are consistent.

The bodies of the files are now presented.

**ecum_init.c**

```
/* AUTOMATICALLY GENERATED FILE. DO NOT EDIT */

/* The EcuM code generator produces this code is response to the
 * list of post build configurable modules in the system.
```

```
 */

/* import configuration index structure
 */
#include "ecum.h"

/* Import prototypes for all init functions needed
 */
#include "xx.h"
#include "yy.h"

void init_modules(void) {
      /* this address is set in the XML to be the start
       * of the configuration data
       * in FLASH. The linker file for pc_ecum.c MUST locate the struct
       * real_configuration_index at this address
       */
      struct configuration_index *index =
            (struct configuration_index *)(0x8000000);

      /* Call all relevant init functions
       */
      xx_init(index->xx_real_configuration);
      yy_init(index->yy_real_configuration);
      /* etc. one line for each module that is post build
       * configurable
       */
}
```

**ecum.h**

```
/* AUTOMATICALLY GENERATED FILE. DO NOT EDIT */

/* This file declares the structure that is the root of
 * all post build configuration data.
 * The members of this struct are generated from the XML as
 * only the XML knows which modules are post build
 * configurable.
 */

struct configuration_index {
      struct xx_configuration *xx_real_configuration;
      struct yy_configuration *yy_real_configuration;
      /* etc. one line for each module that is post build
       * configurable
       */
};
```

**pb_ecum.c**

```
/* AUTOMATICALLY GENERATED FILE. DO NOT EDIT */

/* This file contains the definition of the root of
 * all configuration structures.
 */

#include "ecum.h"
#include "xx_structs.h"
//#include "yy_structs.h"

/* This struct must be loaded at a specific location in memory
 * so that the EcuM can find it.
 */
struct configuration_index real_configuration_index = {
      real_xx_configuration,  /* See xx_pb.c */
      real_yy_configuration,
      /* etc. one line for each module that is post build
       * configurable
       */
};
```

The tooling for this configuration strategy is interesting. It might work as follows:

1. OEM configures post build options in XML.
2. Code generator is run to generate xx_pb.c for each module.
3. All the xx_pb.c modules are compiled.
4. Code generator generates pb_eum.c, ecum.h and ecum_init.c.
5. These are compiled.
6. All the xx_pb.o modules and the ecum module are linked with pb_ecum first in memory.
7. The resulting file is blown into FLASH.

An interesting feature of this scheme is that the linker allocates both FLASH and RAM.

The RAM buffer areas used in these examples and shown in figure 2 have to be allocated with each post build loadable module. This is because the type and amount

of the RAM will not be known earlier. A fixed amount of RAM, therefore, is set aside for buffers, variables, etc. referenced by the xx_pb.c files.

The file xx_pb.c contains the definitions of the RAM for each module. The linker can be directed, via its control file, to place the RAM sections from the xx_pc.o files starting at a known place in RAM. Therefore RAM has been allocated.

> Assumption 4: When allocating RAM in post build loadable and selectable that there is only one contiguous region of RAM that can be used. We do not expect to have to allocate RAM in more than one disjoint region.

It must be understood that a distinction has been made between types types of RAM. The newer type, that is introduced by post build configuration, is allocated at the time the configuration data structures are generated. The older type of RAM is conventionally allocated, either by function calls or at application code link time.

#### 8.4.5.5 Adapting post build for selectable configurations

The implementation of the post build loadable BSW modules does not change for this adaptation. However, the implementation of the EcuM does.

In section 8.4.5.4 the EcuM made use of a single index that contained pointers to each module's configuration. In the post build selectable case there needs to be an array of index structures. The EcuM finds out what the index for the correct configuration is and then finds the pointer to it in the array of indexes.

> Assumption 5: Because the configuration data in FLASH contains pointers to the RAM allocated for that configuration, changing to a different configuration FLASH means that the layout of data in the RAM may also change.

This may introduce problems for external systems (debuggers, etc.) that make assumptions about the layout of this RAM.