**“Schaeffler Automotive Bühl (TF)” C Programming Guidelines in ASW 2019**

**Purpose and Scope**

This document describes the coding standard which must be followed to ensure a high quality and readability code, which is strictly ISO/IEC 9899:1999 compliant and adheres to the implications from the MISRA-C:2012 standards to achieve maximum portability. Furthermore, a corporate and clean coding and commenting style is a prerequisite for presenting the sources to any customer.

The guidelines also cover the aspect of compiler problems by documenting all known issues and the related containments for each target processor, compiler vendor and version used. For being able to write portable, generic and correct code, the required containments on c-code level for all supported compilers shall be respected.

These guidelines are valid for the software writing in the Function and Software Development @ BU eMobility departments (\*) for vehicles in the **ASW 2019** mainline. They have to be used for the writing of new C program code and for the modification of existing C code. Existing software should be adapted when modifications are being made.

These guidelines are also valid when writing simulation code. However, they do not apply to auto-generated code, thought if possible, the tool that generates this code should be “forced” to generated software that closely matches the rules described in this document so that for example, types are consistent between the handwritten code and the generated one.

Though legal aspects (like liability in case of SW failure) and suppliers’ agreement management are beyond the scope of this document these aspects shall be fixed in the contracts made with our customers and suppliers. If Schaeffler integrates code supplied by third party it is recommended that the interfaces shall be compliant to this guideline, which shall be part of the supplier’s agreement.

If Schaeffler code is supplied to any of our customers, the guideline defines the minimum requirements which may be extended by additional rules and standards by the customer. These extensions shall then be documented in the project and communicated to all developers. If third party code supplied by one of our customers or one of the customer’s other suppliers and this code is not integrated into the Schaeffler deliveries but used for internal purpose (e.g. setting up a test harness) this code may be used “as is”.

Contents

[1 Glossary and Definitions 4](#_Toc93508876)

[2 Schaeffler Programming Guidelines in C 5](#_Toc93508877)

[2.1 Applicable standards 5](#_Toc93508878)

[2.2 Naming conventions 5](#_Toc93508879)

[2.3 Variables and constants 6](#_Toc93508880)

[2.3.1 Basic types for variables and constants 6](#_Toc93508881)

[2.3.2 Variables’ storage and usage 6](#_Toc93508882)

[2.3.3 Local variables 8](#_Toc93508883)

[2.3.4 Global variables 9](#_Toc93508884)

[2.3.5 Structured variables 10](#_Toc93508885)

[2.3.5.1 Plain structure 10](#_Toc93508886)

[2.3.5.2 Bit-field structure 12](#_Toc93508887)

[2.3.6 Boolean variables 12](#_Toc93508888)

[2.3.7 Unions 14](#_Toc93508889)

[2.3.8 Arrays 14](#_Toc93508890)

[2.3.8.1 Arrays for different instances 15](#_Toc93508891)

[2.3.9 Interpolation Tables 15](#_Toc93508892)

[2.3.9.1 Maps 15](#_Toc93508893)

[2.3.9.2 Curves 15](#_Toc93508894)

[2.3.9.3 Restriction 15](#_Toc93508895)

[2.3.10 Fixed values 16](#_Toc93508896)

[2.3.11 C-macros 17](#_Toc93508897)

[2.3.12 Enumerator-constants 17](#_Toc93508898)

[2.3.13 Calibration parameters 20](#_Toc93508899)

[2.4 Pointers 21](#_Toc93508900)

[2.4.1 Pointer basics 21](#_Toc93508901)

[2.4.2 Declaration of pointers 23](#_Toc93508902)

[2.4.3 Accessing and modifying pointers 24](#_Toc93508903)

[2.4.4 Memory area location for variable and const pointers 25](#_Toc93508904)

[2.4.4.1 A pointer to a constant 25](#_Toc93508905)

[2.4.4.2 A constant pointer 26](#_Toc93508906)

[2.4.4.3 A constant pointer to constant value 27](#_Toc93508907)

[2.4.4.4 Pointers in structures 29](#_Toc93508908)

[2.5 Preprocessor directives and macros 30](#_Toc93508909)

[2.5.1 Macros 30](#_Toc93508910)

[2.5.2 Function-like macros 32](#_Toc93508911)

[2.5.3 Conditional compilation 32](#_Toc93508912)

[2.5.3.1 Generic code conditional compilation 32](#_Toc93508913)

[2.5.3.2 Project specific code conditional compilation 33](#_Toc93508914)

[2.5.4 Include statements 33](#_Toc93508915)

[2.5.5 Pragmas 34](#_Toc93508916)

[2.6 Functions 35](#_Toc93508917)

[2.6.1 Declaration and Definition 35](#_Toc93508918)

[2.6.2 Parameters and return values 36](#_Toc93508919)

[2.6.3 Termination 37](#_Toc93508920)

[2.6.4 Function calls 37](#_Toc93508921)

[2.6.5 Function pointers 38](#_Toc93508922)

[2.6.6 Passing an array as parameter to a function 38](#_Toc93508923)

[2.7 Operators 40](#_Toc93508924)

[2.8 Control flow 42](#_Toc93508925)

[2.8.1 if 42](#_Toc93508926)

[2.8.2 switch 43](#_Toc93508927)

[2.8.3 for-loop 43](#_Toc93508928)

[2.9 Comments 44](#_Toc93508929)

[2.10 C – Code editor 46](#_Toc93508930)

[2.10.1 Style guide for the Nassi-Shneiderman-Editor (InoSp.exe) 46](#_Toc93508931)

[2.10.1.1 Configuration 46](#_Toc93508932)

[2.10.1.2 Zoom – Status 46](#_Toc93508933)

[2.10.1.3 Structural elements 47](#_Toc93508934)

[2.11 Locator control 48](#_Toc93508935)

[2.12 File names 48](#_Toc93508936)

[3 Known Compiler Problems and Containments 49](#_Toc93508937)

[2 Templates, Checklists, and Tools 50](#_Toc93508938)

[4 References / Further applicable Documents 50](#_Toc93508939)

[5 Change history 50](#_Toc93508940)

# Glossary and Definitions

|  |  |
| --- | --- |
| **Item** | **Description** |
| ISO-C90 | ISO/IEC 9899:1990, Programming languages — C, International Organization for Standardization, 1990 (amended and corrected by ISO/IEC 9899:1990/COR 1:1995, ISO/IEC 9899:1990/AMD 1:1995, and ISO/IEC 9899:1990/COR 2:1996). |
| ISO-C99 | ISO/IEC 9899:1999, Programming languages — C, International Organization for Standardization, 1999 (amended and corrected by ISO/IEC 9899:1999/COR 1:2001, ISO/IEC 9899:1990/COR 2:2004, and ISO/IEC 9899:1999/COR 3:2007). |
| MISRA-C:1998 | MISRA-C:1998 / Guidelines for the Use of the C Language in Vehicle Based Software |
| MISRA-C:2012 | MISRA-C:2012 / Guidelines for the use of the C language in critical systems. |
| “must” / “shall” / “required” | These words are used to define a binding rule. Violations of a binding rule must not be accepted by a reviewer. |
| “may” / “optional” | These words are used to indicate that a rule is truly discretionary. |
| “should” / “recommended” | These words define a recommended practice. If a recommendation is violated the developer shall give a rationale for why the recommendation cannot be followed in this specific case and this has to be accepted by the reviewer. |

# 

# Schaeffler Programming Guidelines in C

## Applicable standards

Any C-code shall be strictly compliant to the ISO-C99 standard including technical corrigendum 1, 2 and 3.

Furthermore, this guideline respects implications coming from the MISRA-C:1998 and MISRA-C:2012 standard, which limits the usage of C language to those elements bindingly defined by the ISO-C99 standard giving no room for platform and compiler specific implementations and hence guaranteeing portability of the code.

These standards however do not cover style or naming conventions (but see below).

Wherever a rule in this guideline is related to the ISO-C or MISRA-C standard this relation will be documented as a rationale.

All source code (including comments) shall be written in British English. This also applies to the syllables and abbreviations used in identifiers.

## Naming conventions

They are described in great detail in the **“S0029\_NamingConventionRules.docx”** companion document.

## Variables and constants

**Note**: In here, the terms definition and declaration occur at several places. In C the difference is as follows:

* **Definition**: The definition creates RAM space, e.g. t\_Si16 Sig\_NEng; outside a block. In this example the definition is also called external definition, since a variable with external linkage is created (the variable can be accessed from “extern”). A variable can be defined only once.
* **Declaration**: The declaration only specifies the type of an object. There can be multiple declarations for the same object.

Each variable shall be defined or declared on a line by its own to improve readability. The intended usage, resolution, ranges and unit shall be documented at definition using comments (for details on commenting style see **§ 2.9**).

|  |  |
| --- | --- |
| **WRONG**: t\_Si8 index, counter; | **CORRECT**: t\_Si8 index;  t\_Si8 counter; |

## Basic types for variables and constants

In order to be independent of the microprocessor or compiler, the basic integer types (signed/unsigned, 8/16/32 bit) as well as floating points ones are defined for each processor/compiler[[1]](#footnote-1),

typedef “32 bit, signed” t\_Si32 ;  
 typedef “16 bit, signed” t\_Si16 ;  
 typedef “8 bit, signed” t\_Si8 ;  
 typedef “32 bit, unsigned” t\_Ui32 ;  
 typedef “16 bit, unsigned” t\_Ui16 ;  
 typedef “8 bit, unsigned” t\_Ui8 ;  
 typedef “bit structure element” t\_UBS ;

typedef “32-bit floating point” t\_F32 ;

Further basic types are linked to the handling of global and local Boolean variables, see **§ 2.3.3** and **§ 2.3.6** for details. All other types (structures, arrays, etc.) are defined only using those processor-independent types. The C predefined types shall not be used.

Note that **TargetLink** also uses the same basic types as its Data Dictionary is generated by JavaICE and thus contains the types defined in the ICE database.

## Variables’ storage and usage

The compiler allows local variables to be of storage class static or auto.

Local auto variables are allocated on the stack and so do not reserve a firm storage location. Their value is at random when the routine is entered, and they have always to be initialized at the routine beginning[[2]](#footnote-2). Therefore, any automatic variable has to be initialized before it is used[[3]](#footnote-3).

To allow for static checking of that rule it is preferred to assign a meaningful value for each of the execution sequences possible instead of assigning (a possibly meaningless) value at definition.

|  |  |
| --- | --- |
| **WRONG**: t\_Si16 trq\_eng\_old = 0; | **CORRECT**: t\_Si16 trq\_eng\_old;  …  trq\_eng\_old = 0; |

Beyond that its address shall not be assigned to another object of longer lifetime, e.g. returned or passed to the calling function[[4]](#footnote-4), as the storage space might be reused for different purpose by the controller, once the function has terminated, and hence the address is meaningless. Accessing the address of an automatic local returned from a subfunction will lead to unpredictable result.

Note: Even though local variables live on the stack their lifetime extends to the execution of the block they are associated to and the call of subfunctions only suspends but does not stop the current blocks execution. I.e. they might exist there during the whole interrupt if they are created within a high level function (e.g. Interupt10ms()).

Be aware, that function parameters behave very similarly to local automatics in the context of a function, despite the fact, that they are initialized to the value passed, when the function is called. This means write access to them is possible, changing its value for all the code after this instruction in the function code. Please refer to **§ 2.6** for more details.

Local static variables “devour” a firm RAM place, are initialized to zero during controller initialization if no explicit initial value is given at definition and contain the last value when the routine is entered again. The initializer for a static variable shall be a constant expression (i.e. no variable)[[5]](#footnote-5).

The access to static variables is typically quicker than on auto-variables but it is recommended to use a global in preference to a local static for several reasons:

* The variables do not show up in the address file or global uniqueness of the identifier name is not enforced by the compiler, which makes it hard or impossible to measure it using a calibration tool.
* A routine that uses static local variables behaves differently when called with the same arguments, as the result depends on the internal state (the history of previous calls to the routine). Furthermore, this hampers unit testing, as internal states cannot be accessed directly and raises the complexity of the test drivers.
* Especially in controller software, static variables are used to remember a value of the last interrupt cycle in order to trigger the change of a signal. Often other routines need to trigger the same event, therefore one global variable for all routines is better (and may also improve clarity).

Variables, that are declared as "static" outside of a routine (only visible to all functions in this file) shall not be used. Those variables must be declared as global.

The default type for variables / parameters / constants shall be "signed", even if there are no negative values (e.g. vehicle speed often is only positive in the software). "unsigned" variables / parameters / constants shall only be used if the range of the signed type is too small and a longer type (t\_Si16 instead of t\_Ui8, or t\_Si32 instead of t\_Ui16) would cost too much RAM or code overhead.

Local variables shall be used, if they are only accessed from within a single function[[6]](#footnote-6). Introduce global variables only if they are indispensable because the access to these variables is possible from everywhere. In the controller software, most variables need to be global to allow visibility and access for the calibration tool. Even though almost all variables are accessible by a routine, there should be a certain discipline when accessing (especially modifying) global variables. This is true especially when global variables from another process are accessed (e.g.: a routine of the shift control accesses a variable that is allocated to the clutch control process). This basic concept of information hiding is supported by the interface model, which can be edited using ICE.

Several variables with the same information shall not exist. This would make the maintenance of the code and its transparency more difficult.

A suitable image must be followed to describe a physical fact with software variables. If physically there can only be one of a selection which could arise (“1 from N”, values/states mutually exclude each other), then one variable has to be declared for its description to which the different values have to be allocated (enumeration).

**Example**: Define a state variable that can have values for either “cold”, “warm”, or “hot”. If instead three bits “s\_Cold\_B”, “s\_Warm\_B”, “s\_Hot\_B” were defined, it would be possible that “s\_Cold\_B” as well as “s\_Hot\_B” are set to true, which is physical nonsense.

## Local variables

Local variables:

* Shall be defined within a function block
* May make use of the following storage classes…
  + **static** in which case we have a variable that is local to that function and persists even when the function has terminated
  + **“none”** in which case we have a variable that is created when the function is called and is lost when the function terminates, is equivalent to specifying no storage class.
* Shall not make use of the following storage class modifiers:
  + **auto**[[7]](#footnote-7)
  + **extern**[[8]](#footnote-8)
  + **register**[[9]](#footnote-9)

Local variables are only visible inside the corresponding {..}-block. Therefore, the variable name **does not need** the “*software component*” suffix, nor any parts of the routine name. Actually, the variable name may be quite short, and only needs to describe the signal/purpose of the variable. Variables which are “only” used for temporary storage (local variables) should have a telling name (“temp\_var”, “temporary\_variable” or something similar would not be very useful).

As local automatics do not need a permanent RAM place, more variables (with a suit­able name in each case) could be declared, instead of re-using the same local variable “temp\_var” for several different intermediate assignments.

EXAMPLE:

|  |
| --- |
| void <Swc>\_Function (void)[[10]](#footnote-10)  {  t\_Si8 idx\_cl; /\* Definition of automatic local variable \*/    static t\_Si16 trq\_cl = K\_ClTrqM\_TrqClMin; /\* Definition and explicit init. \*/  /\* of a static local variable \*/  idx\_cl = IPS\_1; /\* Initialize automatic variable at each call \*/  trq\_cl = TrqCl\_Clh; /\* Initialize static variable at each call \*/  /\* An automatic shall be used in this case. \*/  } |

## Global variables

Global variables…

* Shall be declared in a separate header file[[11]](#footnote-11) that is included by any compilation unit in which they are needed.
* Shall make use of the following storage classes…
  + **extern** in which case we have a variable with external linkage. Note: in C each variable defined outside a block not having the storage class “static” or “register” has the storage class “extern”.
* Shall not make use of the following storage classes…
  + **static** in which case we introduce the concept of local global variables
  + **register**[[12]](#footnote-12)
* It is recommended that there is only a single assignment of global variables per interrupt. For global variables with asynchronous access (e.g. cross interrupt interfaces) this is mandatory to avoid side effects, e.g. toggling values if the value is changed multiple times in a low frequency interrupt task but also read by a higher frequency interrupt task or measurement system. (Measurement systems must use synchronous access.)

Example:

|  |
| --- |
| **header.h** |
| extern t\_Si8 s\_<Swc>\_VariableC; /\* Declaration of variables \*/  extern t\_Si16 s\_<Swc>\_VariableD;  ... |

|  |
| --- |
| **vars.c** |
| #include “header.h” /\*consistency check definition vs. declaration\*/  t\_Si8 s\_<Swc>\_VariableC = 0; /\* Definition of variables \*/  Si16 s\_<Swc>\_VariableD = 0;  ... |

For adaptive values (RAM variables that are buffered in EEPROM during power-off) the same naming-convention as used for global variables applies.

## Structured variables

Structured variables combine variables which belong together. At present two integration criterions are used:

* Derived values from one signal. The “Raw” (=as read in), the “Old”, the “New”, the “filtered”, the “Status” and other values of a signal are integrated in the structure. In this case the structure is named according to the signal (e.g.: NEng.Old, NEng.New).

The recommended standard usage for the New and Old elements of a structure is to assign the “New” value to the “Old” value just before re-calculating the “New” value.

* Integration of variables of one function area. In this case variables with different physical values are integrated in the structure. The name of the structure has to describe the functionality in this case. (E.g.: CoordCl.TrqTarget, CoordCl.SReq, CoordCl.ReqList[],...)

A structure shall not contain members which are considered as output and input to functionality.

A structure shall not contain members that are 2DMap, 3DMap or Curve structures.

When defining a structure type the optional tag shall not be used. An exception to this rule is made for autogenerated code if and only if the tool cannot be configured to omit the tag. In this case the tag has to be a unique identifier[[13]](#footnote-13) (use a “tag” instead of the “t\_” prefix).

RECOMMENDED:

typedef struct {

…

} t\_<ID> ;

TOLERATED (autocode only):

typedef struct tag<ID> {

…

} t\_<ID> ;

NOT ALLOWED:

typedef struct t\_<ID> {

…

} t\_<ID> ;

There are two possible ways to create a structured variable: use a **plain structure** or a **bit-field structure**. Each structure shall be limited to at most 1023 elements, whatever their size[[14]](#footnote-14).

## Plain structure

In that case, the structure members (for example Old, New, etc.) can be of any type. This is the most versatile construct but the one that will take more space in memory.

**Example**:

|  |
| --- |
| typedef struct {  t\_Ips New;  t\_Ips Old;  } t\_IpsNO ; |

**Memory used by a plain structure**

At first sight, a plain structure size is equal to the size of all its members. Thus, if a structure is made of a byte and a short (two bytes), then the structure should be three bytes long. In practice however, a structure size can be bigger than the sum of the size of its members because of optimization considerations (less access cycles) or hardware architecture restrictions.

As a direct consequence, a structure member of size N bytes can be accessed only if it is aligned on an address that is a multiple of N. Thus, a byte can be stored at any address, a short only at addresses multiple of two, etc.

Let’s imagine that a structure over the course of the time has been defined as follows (at the beginning, it has only a t\_Si8 and a t\_Si32 member and then a new t\_Si8 and a t\_Si16 member was added later on):

typedef struct {

t\_Si8 FirstMember;

t\_Si32 SecondMember;

t\_Si8 ThirdMember;

t\_Si16 FourthMember;

} typEvolvingStructureOverTime ;

Now let’s suppose a variable called MyStruct of that type is stored at address 0x00000000. In that case, the structure is represented in memory as follows (on a little endian architecture[[15]](#footnote-15)):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0x00000000** | **0x00000001** | **0x00000002** | **0x00000003** |
| **0x00000000** | FirstMember |  |  |  |
| **0x00000004** | SecondMember | | | |
| **0x00000008** | ThirdMember |  | FourthMember | |

Thus, while one might expect a size of 8 bytes for this variable (1+4+1+2), sizeof(MyStruct) will in fact return 12 or 50% more!

If this variable is now an array of for example 200 elements, then instead of taking 1600 bytes in memory, it will take 2400 bytes, which is definitively a huge waste of memory (+ 800 bytes).

Let’s rewrite this structure definition as follows:

typedef struct {

t\_Si8 FirstMember;

t\_Si8 ThirdMember;

t\_Si16 FourthMember;

t\_Si32 SecondMember;

} typEvolvingStructureOverTime ;

Then, the structure members are now mapped in memory as follows (on a little endian architecture):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0x00000000** | **0x00000001** | **0x00000002** | **0x00000003** |
| **0x00000000** | FirstMember | ThirdMember | FourthMember | |
| **0x00000004** | SecondMember | | | |

And sizeof(MyStruct) returns 8, which is indeed the expected size.

Of course, depending on the member types, you may get a less favourable result. But still, when paying attention to their alignments and reordering them in a more sensible way (typically by grouping them by type), you will always get a better size for the structure and hence you will reduce the amount of lost memory.

## Bit-field structure

In that case, each structure member is declared as having a length of X bits. This is done by appending the “:X” syntax at the member end. The type of each member must be*unsigned int*[[16]](#footnote-16). However, in order to be independent of the microprocessor or compiler and to allow clear distinction in the code, bit-field elements shall be of type **t\_UBS**, whatever the value of X.

**Example**:

|  |  |
| --- | --- |
| typedef struct {  **t\_UBS** Source : 2 ; // len 2  **t\_UBS** DataValid : 2 ;  **t\_UBS** SafeEepromStatus : 2 ;  } t\_Tcu\_NvMemStatus\_BS; | typedef struct {  **t\_UBS** IgnOn\_B : 1 ; // len 1  **t\_UBS** EngOn\_B : 1 ;  } t\_Cif\_Cmn\_BS; |

A bit-filed structure uses much less memory than a plain structure but requires more code instructions. For example, t\_Tcu\_NvMemStatus\_BS fits into one byte, while an equivalent plain structure would need at least three bytes (one byte for each member).

A structure which exclusively consists of one or more bitfields is called a **bitset**.[[17]](#footnote-17)

## Boolean variables

Terminology:

* **Flag**. An object that can either be TRUE or FALSE and is represented by one bit. Can be implemented as typUBS of length 1.
* **Boolean**. An object that can have one of two states. It can be represented by one or more bits. A Flag is a Boolean. A Boolean can be represented by a variable which has only two allowed values.

The ISO-C 90 standard does not define a Boolean type. The only Boolean value in C is the result of a relational operator, which is defined to be of type signed int[[18]](#footnote-18).

Boolean variables shall be implemented as follows (see also example below):

* The default solution is to define them in a structure of bit-fields of length 1.
* If Boolean variables need to be defined for multiple objects/cases, the preferred solution is to define such a variable only once as a bit-field element in a structure, and then define an array of this structure type, with one index for each case.
* In cases where processing time is critical and controller storage capacity sufficient or for local Boolean values, it is permitted to use one or more variables of t\_Boolean. It is still recommended to keep the Boolean variables in a structure (using elements of t\_Boolean instead of t\_UBS of length one).
* The structure implementation shall be visible when accessing the Boolean variables, i.e. no macros shall be used to hide the implementation, as this makes debugging more complex.

**Example #1**: Structure of Boolean variables represented by bit-field members of length one.

Suppose we want to define a set of Boolean variables for a SW component. This is done by declaring each of these as a bit-field of length one bit in a structure.

typedef struct {

t\_UBS CloseCmd\_B : 1;

t\_UBS OpenCmd\_B : 1;

} t\_Swc\_Flags\_BS;

Define the variable s\_Swc\_Flags\_BS…

t\_Swc\_Flags\_BS s\_Swc\_Flags\_BS;

…and the usage of the Boolean variable within the source code looks like:

s\_Swc\_Flags\_BS.CloseCmd\_B = TRUE;

To duplicate the Boolean variables for two input shafts, define an array of type t\_Swc\_Flags\_BS.

Note that it is NOT allowed to define an array directly of the bit-field elements. Hence, it is recommended to create separate structures for bit-field elements which are duplicated for two input shafts and bit-field elements which need no duplication.

Define the variable s\_Swc\_FlagsAt\_BS …

t\_Swc\_Flags\_BS s\_Swc\_FlagsAt\_BS[IPS\_NUM];

…and the usage of the Booleans is as follows:

for ( ips = IPS\_1; ips < IPS\_NUM ; ips++ ) {

s\_Swc\_FlagsAt\_BS[ips].CloseCmd\_B = TRUE;

}

**Example #2**: Structure of Boolean variables, represented by t\_Boolean

typedef struct {

t\_Boolean EngineRunning\_B;

t\_Boolean EngineCtrlInitialised\_B;

} t\_Sig\_EngineStatus;

t\_Sig\_EngineStatus s\_Sig\_EngineStatus;

t\_Boolean engine\_running\_b = s\_Sig\_EngineStatus.EngineRunning\_B;

Boolean variables (i.e. bit-field members of length one or typBoolean entities) shall only be assigned to "TRUE", "FALSE", another Boolean identifier or a **Boolean expression**[[19]](#footnote-19). Boolean variables shall only and explicitly be compared to TRUE or FALSE.

**Definition**: an *expression* is anything which needs to be evaluated, a condition, validity of an array, validity of memory, etc. When the expression evaluated to only and only two values 0 (FALSE) or 1 (TRUE), then it is called a *Boolean expression*.

The bellow examples are Boolean expressions:

( a == b )

( (a == b) && (c == d) )

( (a <= b) || ((c <= K\_x\_y) && (a <= K\_y\_z)) || ((A >= T.Old) && (c > 0)) )

Thus, theoretically, it is possible to write something like this:

s\_Swc\_Flags\_BS.CloseCmd\_B = ( (a <= b) || ((c <= K\_x\_y) && (a <= K\_y\_z)) || ((A >= T.Old) && (c > 0)) ) ;

However, please do not use such a construct. Not only this does not help to achieve more readable code, but this also increases the HIS.VOCF metric[[20]](#footnote-20). In fact, the TF-UnallowedConstruct custom Axivion script will report this code as a violation.

To sum up, Boolean expressions can be used, but they must not contain more than three Boolean conditions (this is enforced by the TF-UnallowedConstruct custom Axivion script).

**Example #3**: Below are acceptable Boolean expressions:

entry\_gear\_and\_neng\_b = ( (exit\_gear\_and\_neng\_b == FALSE)

&& (s\_Psig\_NEng.New < n\_eng\_max\_cl\_close\_suggest) );

s\_Swc\_CoordClCloseStart\_BS[idx\_ips].RequestCloseClExist\_B = \

( s\_Swc\_ClOpenClosePrioAt[idx\_ips].Close >= SWC\_OCPRIO\_WHEN\_POSSIBLE );

The names for the bit variables (flags) must be chosen so that it is clear when they are TRUE (the bit is set to 1). The flag “b\_gear\_shift\_direction” does not specify its state when upshifting. The flag called: “b\_gear\_upshift” would be more telling!

In the control software there shall be a positive logic for all bit variables. For example, if the bit s\_Sig\_Flags\_BS.EngRunning\_B is set to TRUE (which is 1), then the engine is on, while with s\_Sig\_Flags\_BS.EngRunning\_B == FALSE (the bit is 0), the engine is off.

## Unions

Unions shall not be used due to compatibility issues[[21]](#footnote-21).

## Arrays

The definition of the length of an array must always use an explicit #define value. It is not allowed to use a calculation of different #defines or #defines + numerical values. Background for this rule is the generation of the A2L file from ICE (Hexwork currently does not interpret arithmetic operations).

WRONG: t\_Ui8 MyEmots[NUM\_GB\_EMOT + NUM\_CL\_EMOT]  
 t\_Ui8 MyEmots[NUM\_GB\_EMOT + 1] ;

CORRECT: #define NUM\_EMOT NUM\_GB\_EMOT+NUM\_CL\_EMOT

t\_Ui8 MyEmots[NUM\_EMOT] ;

## Arrays for different instances

Arrays keeping the values for the (two) input shafts or two different clutch actuators shall be named with an “At”, like following examples:

t\_Swc\_Gear s\_Swc\_GearAt[NUM\_IPS] ;

t\_Swc\_StatusInt\_BS s\_Swc\_StatusIntAt\_BS[NUM\_CA] ;

In general, all array keeping data for different instances of a functionality unit, shall follow this naming convention. Other arrays shall not use the “At”.

## Interpolation Tables

For the definition of interpolation tables, axis and value arrays are needed. In most cases these axis and value arrays are only used for the definition of the interpolation table.

See **01\_swc\_components/arc/03\_docu/S0029\_NamingConventionRules.docx** for more information.

## Maps

APPCONST t\_Si8 ca\_Swc\_DeltaAVehHillGr[8] = {-10, -4, -1, 0, 1, 2, 10, 20};

APPCONST t\_Si8 ca\_Swc\_NInpActGr[5] = {-10, 20, 40, 80, 100};

APPCONST t\_Si8 cm\_Swc\_TrqSumIncMaxGr[5][8]= {  
 { 5,   5,   8,   8,   8,   12,   25,   20 },  
 { 5,   5,   8,   8,   8,   12,   25,   20 },  
 { 5,   5,   7,   7,   7,   10,   20,   20 },  
 { 5,   5,   5,   5,   5,   10,   20,   20 },  
 { 5,   5,   5,   5,   5,    8,   20,   20 }  
 };

APPCONST t\_Si8Map2D **c\_**Swc\_TrqSumIncMaxGr = {  
 8,  
 5,  
 &**ca\_**Swc\_DeltaAVehHillGr1[0],  
 &**ca\_**Swc\_NIpsActGr1[0],  
 &**cm\_**Swc\_TrqSumIncMaxGr1[0][0]  
 };

## Curves

Same logic, but cv\_ must be used instead of cm\_ for the data array:

APPCONST typSI16Curve c\_Colu\_LimitVolumeFlow = {

COLU\_NR\_LIMIT\_VOL\_FLOW\_TEMP\_VRTX,

&**ca**\_Colu\_LimitVolumeFlowTemp[0],

&**cv**\_Colu\_LimitVolumeFlowVa[0]

};

## Restriction

A 2DMap, 3DMap or Curve interpolation table structure shall not be used a member of another structure.

## Fixed values

There are different types of fixed values in the software:

* Numbers that are fixed for a project and cannot be changed without changing the associated code, i.e. are only precompilation configurable (e.g. array sizes, variable ranges, etc.): These Numbers will get a symbolic name using C-macro, since the final value is needed at compilation time. For additional information on C-macros and their intended usage please refer to **§ 2.5.1**.
* Numbers that describe the state of a state machine, or status of a sensor, etc.: These numbers will get a symbolic name using C-macro or an **enumeration type**.
* Values that are fixed but may be project specific. These values also referred to as calibration parameters (as they can be made post-build configurable using a calibration tool) must not be represented by a C-macro but by a constant, as this does not allow for separation of code and data.

There shall be no fixed numerical value in the code. These “magic numbers” should be replaced by C-macros or enumerator-constants with meaningful names (plus comments). The symbolic name must not reflect the numerical value, but the purpose/functional meaning.

The only allowed exceptions are numbers that are intrinsic to a formula or the initialization of a loop counter with 0 or 1. This also applies to conversion factors that are defined by physics and therefore will never change (e.g. rpm -> rad) as well as multiplication or division by multiple of 10 (10, 100, 1000, etc.) The usage of magic numbers shall be documented in detail in the code.

Examples:

ALLOWED: average = (a + b) / **2**;

for( i = **0** ; i < MAX\_ELEM ; i++ )

## C-macros

In order to use #defines together with autocode generation data dictionaries, #defines for numeric values must have a typecast to specify the type of data.

WRONG: #define GEAR\_5 5

CORRECT: #define GEAR\_5 ((t\_Gear)5)

Where possible avoid using #define within another #define, as the target link dictionary does not support this:

WRONG:  
 #define ELEMENTS\_IN\_GEAR\_ARRAY ((t\_Si8)(NUM\_GEARS + 1))

Function like #defines shall be declared as c-functions instead of #defines:

WRONG:  
 #define GET\_UI8\_0X00\_OF\_32BIT(x) ((t\_Si8)( ( (t\_Ui32)(x) ) >> 16 ) ))

Comments shall not be added inside the #define and correspondingly comments in ICE shall only be added in the description field to avoid the generation of the comment in the same line as the #define.

WRONG: #define SI8\_MIN ((t\_Si8) (-128)) /\* signed 8-bit min value \*/

CORRECT: /\* signed 8-bit min value \*/  
 #define SI8\_MIN ((t\_Si8) (-128))

When #defines are related to each other’s, it is recommended that they all start with the same prefix. This will make the code much more readable and searchable.

**Example**:

#define SWC\_TASK\_OUT\_CL\_CLOSE\_CMD ((t\_Ui8)0x04)

#define SWC\_TASK\_OUT\_CL\_OPEN\_CMD ((t\_Ui8)0x02)

#define SWC\_TASK\_OUT\_POWER\_ONOFF ((t\_Ui8)0x20)

However, you may consider using enumerator-constants instead (see below).

**Note**: Integer suffixes such as ‘l/L’ or ‘u/U’ shall not be used. Use a typecast instead.

WRONG: #define IPS\_SPEED\_DIFFERENCE\_FACTOR 50L

CORRECT: #define IPS\_SPEED\_DIFFERENCE\_FACTOR ((t\_Si32) 50)

## Enumerator-constants

Enumerator-constants are constants that are regrouped within an enumerator type. They allow grouping related constants. The list of enumerator-constants with an enumerator type is called an enumerator list. Enumerator-constants are particularly useful when defining constants with consecutive values as the compiler automatically adds one to the next enumerator-constant in the list.

By default, the first enumerator-constant is initialized to 0, but it is possible to change this value with an explicit initialization (using the ‘=’ construct). This initialization construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized[[22]](#footnote-22).

Please note that it is perfectly possible to have different enumerator-constants initialized with the same value.

Because they are related to each other, each member of an enumeror list must start with the same prefix (**TF Naming Rule 11.11**).

**Example #1**:

typedef enum {

<SWC>\_CLOSE\_ALLOW\_NONE,

<SWC>\_CLOSE\_ALLOW\_INACTIVE,

<SWC>\_CLOSE\_ALLOW\_ACTIVE,

<SWC>\_CLOSE\_ALLOW\_BOTH

} t\_<Swc>\_IpsClCloseAllow;

The t\_Swc\_IpsClCloseAllow enumerator type defines four enumerator-constants. Because there is no initialization construct, this means <SWC>\_CLOSE\_ALLOW\_NONE has a value of 0. As the other values are atomically increased by one, this means <SWC>\_CLOSE\_ALLOW\_INACTIVE = 1, <SWC\_CLOSE\_ALLOW\_ACTIVE> = 2, etc.

**Example #2**:

The TASK\_OUT… macros could be put into an enumerator type as follows:

typedef enum {

<SWC>\_TASK\_OUT\_CL\_CLOSE\_CMD = 0x04,

<SWC>\_TASK\_OUT\_CL\_OPEN\_CMD = 0x02,

<SWC>\_TASK\_OUT\_POWER\_ONOFF = 0x20

} t\_<Swc>\_TaskOut;

**Example #3**:

typedef enum {

IPS\_1 = 0,

IPS\_2 = 1,

IPS\_NUM = 2,

IPS\_CII\_NUM = 2,

IPS\_UNKNOWN = 2

} t\_Ips;

In this example, IPS\_NUM, IPS\_CII\_NUM and IPS\_UNKNOWN have the same value (2).

With most compilers, using an enum will take up no space on the target, will be optimized just as well as if you had used a #define, and has the added benefit of better debug information (they are visible in a A2L file for example). They have some limitations though:

1. An enumerator-constant has always the type **int**[[23]](#footnote-23). This means that when used in arithmetic expressions with unsigned numbers, a typecast is needed to avoid getting Misra-C 2012 10.3, 10.4, 10.6 or 10.7 violations.
2. Unless a specific compiler option is used to force enums to always be represented as int, the size of an enumerator type is unknown (this is compiler dependant). Thus trying to coerce an enumerator type to another type using a pointer to extract part of that type may lead to incorrect behaviors.

**Example**:

typedef enum

{

ENUM\_FIRST = 0x01,

…

ENUM\_LAST = 0xFF,

} t\_SomeEnum;

t\_SomeEnum s\_SomeEnum = ENUM\_LAST ;

Because you know that your enum fits into a byte, you may be tempted to write the following code:

void SendToCan(t\_Ui8 \*ptr\_to\_can)

{

can\_buffer[0] = ptr\_to\_can[0] ;

}

…

SendToCan((t\_Ui8 \*) &SomeEnum) ;

However, because the 0xFF value does not fit into a signed char (a default size for an enum), a compiler may decide to use a signed int instead to represent the s\_SomeEnum enum.

Let’s assume a signed int on a specific target processor is 32 bits width and the architecture is little endian. In that case, SomeEnum is represented in memory as follows (for the sake of simplicity, we assume it is stored at address 0x00000000):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0x00000000** | **0x00000001** | **0x00000002** | **0x00000003** |
| **0x00000000** | 0xFF | 0x00 | 0x00 | 0x00 |
| **0x00000004** |  |  |  |  |

In the above code, ptr\_to\_can will have adresse 0x00000000 and ptr\_to\_can[0] will return 0xFF. Thus the code works as expected.

On the other hand, if the targer processor has a big endian architecture, SomeEnum is represented in memory as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0x00000000** | **0x00000001** | **0x00000002** | **0x00000003** |
| **0x00000000** | 0x00 | 0x00 | 0x00 | 0xFF |
| **0x00000004** |  |  |  |  |

As ptr\_to\_can still has address 0x00000000, then ptr\_to\_can[0] will always return 0x00, whatever the value assigned to SomeEnum. This is definitively not what you want.

To sum up, do not try to use enums with pointers.

The correct way to handle an enum in that case is to use an intermediate variable of the expected size:

t\_Ui8 some\_data ;

some\_data = (t\_Ui8) SomeEnum ;

SendToCan(&some\_data) ;

## Calibration parameters

Calibration constants shall be used with external linkage only to ensure they are accessed by reference (their address in memory) and not by their value (directly compiled into the assembler code), otherwise a calibration tool will fail to access and modify the value. This is achieved by defining all constants in separate calibration data files and nowhere else and treating these files as a separate translation unit. A constant shall be defined in the calibration data file of the process that owns it.

Braces shall be used to indicate and match the structure in the non-zero initialization of array and structure type constants[[24]](#footnote-24). This allows the compiler to check the initializers for correct structure when compared to the type of the constant.

WRONG:

const t\_Si16 c\_Swc\_SomeId[2][3] = {1, 2, 3, 4, 5, 6};

CORRECT:

const t\_Si16 c\_Swc\_SomeId[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

Const shall not be used together with typedef.

WRONG:

typedef const t\_Si16 t\_SomeConstant;

t\_SomeConstant c\_Swc\_MyConst = 5;

CORRECT:

const t\_Si16 c\_Swc\_MyConst = 5;

The type qualifier volatile shall not be used for declaration or definition of constants.

Octal constants (i.e. a number with a leading zero like 012, which is 10d but could be easily misunderstood as 12d) shall not be used[[25]](#footnote-25).

## Pointers

## Pointer basics

Generally speaking, a pointer in programming is a data type, whose value refers to (points to) another value stored elsewhere in the physical memory by using its address. A pointer in C holds the address of an object or function in memory, i.e. it is a reference to that object, which is why deriving the address of an object is also called “referencing” and obtaining the object from the address the pointer refers to is called “dereferencing”. As pointers in C allow unprotected access to memory and the standard checking of C-compilers is rather weak due to the limitations of the language, usage of pointers increases the risk of severe programming error.

The basic operators related to pointers in the ISO-C Standard[[26]](#footnote-26) are the unary operators “address of” (&) and “indirection” (\*).

The “address of” operator applied to a function or object returns a pointer to that function (the function’s entry point in memory to be more precise) or object. If the object has the type “type” the result has the type “pointer to type”. One can recursively apply referencing to the address of an object (the pointer to that object) resulting in a “pointer to a pointer to type” and so on. In C each “pointer to” is referred to as one “level of pointer indirection”, i.e. the address of operator adds one “level of pointer indirection”.

The operand of the “indirection” operator shall have pointer type and the result of applying it to a “pointer to function” or a “pointer to object” if the function or object, respectively. The “indirection” operator removes one “level of pointer indirection”.

**Example**:

t\_Ui8 \*var\_ptr;

t\_Ui8 var;

Declares var\_ptr as pointer to t\_Ui8 and a variable var of type t\_Ui8.

var\_ptr = &var;

Retrieves the memory address of var and assign it to the pointer and

\*var\_ptr = 0xFF;

Modifies the value of var via the pointer.

To explain the results on controller level let’s assume to have a controller using 32 Bit addresses with a minimum addressable unit (MAU) of 1 Byte and let’s assume the pointer “var\_ptr” is stored as a 32 Bit address at address 0x00001004 and the variable of t\_Ui8 is stored as a Byte at address 0x0000100D. Then the resulting memory content is as follows



There is no explicit pointer type defined in the ISO-C Standard. A pointer to type holds the address of the object in memory it refers to, i.e. its size depends on the data size of an address of the hardware environment used. A “pointer to t\_Ui8” in an environment with 32 Bit addresses for example is 32 Bits wide. Therefore, a cast shall not be performed between a pointer type and an integral type[[27]](#footnote-27).

Other operators related to pointers are the “array index” operator ([]) and the “indirect member access” operator (->).

The “array index” operator is related to pointers, as arrays in C can be thought of “pointers to contiguous areas in memory”. By declaring an array, the identifier representing the array is in fact the “pointer to the first element of the array”.

**Example**:

t\_Si16 array[5];

Declares array[5] to be an array of five elements of type t\_Si16, while array itself is a pointer to the first element, i.e. &array[0]. So, applying the “array index” operator to an array implicitly includes dereferencing the pointer to that element.

C also defines, that array indexing must be equivalent to the pointer arithmetic related to them, which makes the following lines identical

array[i]

\*(array + i)

This can be misleading, because the address is not incremented by i, as it could be expected from the statement, but by i times the size of the values in the array in memory. In the example above, if we assume the address of the first element “&array[0]” is 0x1000 and the size of t\_Si16 is two bytes, then array + i yields (0x1000 + 2\*i). Therefore, pointer arithmetic’s shall not be used.

When declaring a pointer to a structure type:

typedef struct {

t\_Si16 New;

t\_Si16 Old;

} t\_Si16NO;

t\_Si16NO \*struct\_ptr;

Applying the “indirect member access” operator to the “pointer to structure” is equivalent to applying the “direct member access” operator (.) to the dereferenced “pointer to structure”, i.e. it implicitly includes dereferencing:

struct\_ptr->New

Is the same as below:

(\*struct\_ptr).New

As using the “indirect member access” operator with “pointer to structures” improves readability, it shall be used in preference to the alternative operation.

## Declaration of pointers

Definitions on declaration, especially declaration of pointers, are rather abstract in ISO-C standard[[28]](#footnote-28). For the explanation of the basics within this document let’s look at a simplified syntax description for a declaration

storage\_class\_specifier type\_qualifier type\_specifier declarator

A storage class specifier for pointers includes extern, static and auto, while the type qualifier can be const or volatile. As there is no explicit pointer type defined by the ISO-C standard, the type specifier can only be one of the standard types[[29]](#footnote-29) defined in this document. To declare a pointer the “indirection” operator (\*) is added to the identifier used in the declarator.

**Example**:

t\_Si16 \*example\_ptr;

Declares example\_ptr as “pointer to t\_Si16”, i.e. the value stored in example\_ptr is meant to be the address of a t\_Si16 value. To understand the syntax, it may help to think of the example being equivalent to:

t\_Si16 (\*example\_ptr);

I.e. the dereferenced example\_ptr is declared to be a t\_Si16 value.

When using a type qualifier in the declaration:

const t\_Si16 \*to\_const\_ptr;

to\_const\_ptr is declared to be a “*modifiable pointer to a constant t\_Si16 value*”, which means, that the value pointed to cannot be modified via the pointer (this is checked at compiler time), whereas the pointer itself (the address) can be changed. A type cast shall never be used to remove such qualification from the type addressed by a pointer[[30]](#footnote-30).

In a more complex example, the declarator itself can also contain type qualifiers:

t\_Si16 \*const const\_ptr;

Declaring const\_ptr to be “*constant pointer to a modifiable value*”, which means that the value pointed to can be modified via the pointer, but the pointer itself cannot be changed.

If one needs to use a type name for a pointer type, e.g. for a type-cast, the standard defines it to be a declaration omitting the identifier.

**Example**:

t\_Si16 \*

Is a “pointer to t\_Si16” type which can be used for casting any “pointer type” to “pointer to t\_Si16”

example\_ptr = (t\_Si16 \*) something\_ptr;

It is recommended to use only one level of pointer indirection and more than two levels of indirection shall not be used[[31]](#footnote-31), as this can seriously impair the ability to understand the code.

**Example**:

CORRECT: t\_Si16 \*s\_Swc\_TrqEng\_Ptr;

t\_Si16 \*s\_Swc\_TrqEngPtr\_Ptr;

WRONG: t\_Si16 \*\*\* s\_Swc\_TrqEngPtrPtr\_Ptr ;

t\_Si16 \*\* s\_Swc\_NIpsPtrPtrAt[NUM\_IPS]

s\_Swc\_TrqEng\_Ptr is a “*pointer to t\_Si16*”, s\_Swc\_TrqEngPtr\_Ptr is a “*pointer to a pointer to t\_Si16*” and s\_Swc\_TrqEngPtrPtr\_Ptr is a “*pointer to a pointer to a pointer to t\_Si16*”, which is three levels and therefore illegal.

s\_Swc\_NIpsPtrPtrAt is also illegal, because arrays are implicitly converted to “*pointers to the first array elemen*t”, which adds an extra level of pointer indirection here.

## Accessing and modifying pointers

Pointer arithmetic shall not be used. This rule is relaxed for pointers addressing arrays or array elements, for which array indexing is the only allowed form of pointer arithmetic[[32]](#footnote-32).

**Example**:

void CalcSomething( t\_Si16 \*param\_ptr, t\_Si16 param\_array[])

{

param\_ptr = param\_array; /\* COMPLIANT \*/

param\_ptr ++; /\* ILLEGAL: pnt. increment \*/

param\_ptr = param\_array + 5; /\* ILLEGAL: pnt. arithmetic \*/

param\_ptr = &param\_array[5]; /\* COMPLIANT \*/

param\_ptr = &param\_ptr[5]; /\* ILLEGAL: no pnt. to array \*/

param\_array[5] = 0; /\* COMPLIANT \*/

}

At hardware level, a certain memory alignment is related to each data type, e.g. 16-bit values can only be stored at even addresses. By using pointer and casting them to pointers of different type one can easily violate these alignment rules, resulting in unpredictable behaviour.

Let’s assume we have a controller with 32-bit addresses that can store values wider than 8 Bits on even addresses only. Let’s also assume there is an array of 4 t\_Si8 values called array and that covers a contiguous area of Bytes in memory, starting at address 0x0000100C and a pointer to t\_Si16 called val\_ptr and stored at address 0x00001004.

At last, let’s retrieve the address of the array second element array[1] (which is stored at odd address 0x0000100D) and assign it to the pointer.

Then, accessing the value referenced by the pointer leads to accessing a 16-bit value at an odd address, which will cause an incorrect memory access alignment exception on most controllers.

**Example**:

t\_Si8 array[4];

t\_Si16 \*val\_ptr;

val\_ptr = (t\_Si16 \*)(&array[1]);

\*val\_ptr = 0x1234;



Therefore, it is not recommended to cast a pointer to object type to any other pointer to object type, and such cast is forbidden, if the new pointer type requires a stricter alignment[[33]](#footnote-33).

## Memory area location for variable and const pointers

In embedded systems the memory is normally separated into ROM and RAM memory. Modifiable variables or pointers are stored in RAM. Global constants and calibration data are stored within the ROM (mostly a flashable memory).

Constant pointers shall also be located in the ROM, therefore the differences between the following pointer declarations that all use a const must be clarified. The focus is only on global constants / pointers as for these the location in ROM / RAM is relevant.

* pointer to a constant value
* constant pointer to a variable value
* constant pointer to constant value

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Declaration** | **pointer** | **element pointed to** | **element modifiable via pointer** | **Memory area** | |
| **Pointer** | **Element** |
| **TYPE \* s\_ToVariable\_Ptr;** | modifiable | variable | yes | RAM | RAM |
| **const TYPE \* s\_ToConstVariable\_Ptr;** | modifiable | variable (treated as const) | no | RAM | RAM (not modifiable via pointer) |
| const | no | RAM | ROM |
| **TYPE \* const c\_<Swc>\_ToVariableConst\_Ptr;** | const pointer | variable | yes | ROM | RAM |
| **const TYPE \* const c\_<Swc>\_ToConstVariableConst\_Ptr;** | const pointer | variable (treated as const) | no | ROM | RAM (not modifiable via pointer) |
| const | no | ROM | ROM |

## A pointer to a constant

* is declared as

const TYPE \* s\_ToConstConstVariable\_Ptr;

* is located in RAM, the pointer itself is NOT const
* can be modified to point to a different constant
* can be modified to point to a variable, but the content pointed to **cannot** be modified



Be aware that the examples use Big Endian representation. In case of Little Endian the representation needs to be changed correspondingly.

**Example**:

/\* Calibration constants that cannot be modified and are located in **ROM** \*/

const t\_Si16 c\_Val1 = 0x0815;

const t\_Si16 c\_Val2 = 0x4711;

/\* variable – located in **RAM** \*/

t\_Si16 s\_var\_1 = 0x2412;

/\* pointer to a constant located in RAM \*/

**const t\_Si16 \* s\_ToConstVariable\_Ptr;**

/\* pointer can be modified to point to a different constant \*/

to\_const\_ptr = &c\_Val1;

to\_const\_ptr = &c\_Val2;

/\* pointer can be modified to point to a variable,

/\* but **variable cannot be modified via the pointer** \*/

s\_ToConstVariable\_Ptr = &s\_var\_1;

\* s\_ToConstVariable\_Ptr = 9876; /\* 🡪 **ERROR**: pointing to const object \*/

## A constant pointer

* is declared as

TYPE \* const c\_<Swc>\_ToVariableConst\_Ptr;

* is located in ROM
* cannot be modified
* can point to a variable
* content of variable can be modified via the constant pointer



**Example**:

/\* global constants that cannot be modified and are located in **ROM** \*/

const t\_Si16 c\_Val1 = 0x0815;

const t\_Si16 c\_Val2 = 0x4711;

/\* variable – located in **RAM** \*/

t\_Si16 s\_var\_1 = 0x2412;

/\* constant pointers to a fixed address \*/

/\* ptr in ROM, address pointed to in ROM \*/

t\_Si16 \* const c\_<Swc>\_ToVariableConst1\_Ptr = &c\_Val2;

/\* 🡪 **Compiler** **Warning**: “different const qualifier” \*/

/\* **MUST NOT BE USED!** \*/

/\* ptr in **ROM**, variable pointed to in **RAM** \*/

**t\_Si16 \* const c\_<Swc>\_ToVariableConst2\_Ptr = &s\_var\_1;**

/\* variable pointed to by a const pointer \*/

/\* can be modified directly or via pointer \*/

s\_var\_1 = 12345; /\* OK \*/

\*c\_<Swc>\_ToVariableConst1\_Ptr = 43981; /\* OK \*/

## A constant pointer to constant value

* is declared as

const TYPE \* const **c\_<Swc>\_ToConstVariableConst\_Ptr**;

* is located in ROM
* pointer and memory content it points to cannot be modified
* can point to a constant
* can point to a variable
* content pointed to **cannot** be modified



**Example**:

/\* global constant that cannot be modified and is located in **ROM** \*/

const t\_Si16 c\_Val1 = 0x0815;

/\* variable – located in **RAM** \*/

t\_Si16 s\_var\_1 = 0x2412;

/\* constant pointer to a constant \*/

/\* ptr in **ROM**, address pointed to in **ROM** \*/

const t\_Si16 \* const c\_Swc\_ToConstVariableConst1\_Ptr = &c\_Val1;

/\* pointer can point to a variable or an address in RAM, \*/

/\* but **var\_1 cannot be modified** via pointer \*/

const t\_Si16 \* const c\_Swc\_ToConstVariableConst2\_Ptr = &s\_var\_1;

/\* after declaration pointer cannot be modified to point \*/

/\* to a different address (neither in ROM nor in RAM) \*/

c\_Swc\_ToConstVariableConst1\_Ptr = &c\_Val1; /\* **ERROR** \*/

/\* variable cannot be modified via pointer \*/

\* c\_Swc\_ToConstVariableConst2\_Ptr = 3456; /\* **ERROR** \*/

## Pointers in structures

A constant of a structure type is only constant if all structure members are constant. Therefore, the pointers in the structure must be declared as a “constant pointer to a constant value”. Then the constant is located in ROM.

**Example**:

typedef struct {

**const** t\_Si8 \* **const** TableIn\_Ptr;

**const** t\_Si8 \* **const** TableOut\_Ptr;

} t\_ConfigTables;

This constant is located in ROM:

const t\_Si8 c\_<Swc>\_TableIn[…];

const t\_Si8 c\_<Swc>\_TableOut[…] ;

t\_ConfigTables c\_Swc\_Structure = {

c\_<Swc>\_TableIn, /\* in ROM \*/

c\_<Swc>\_TableOut, /\* in ROM \*/

};

This one is not:

const t\_Si8 c\_<Scw>\_TableIn[…];

t\_Si8 s\_TableOut[…] ;

typConfigTables c\_<Swc>\_Structure = {

c\_<Swc>\_TableIn, /\* in ROM \*/

s\_TableOut, /\* in RAM \*/

};

## Preprocessor directives and macros

## Macros

C-macros are replaced by their definition (expanded) by the preprocessor before compilation. Macros shall only expand to[[34]](#footnote-34):

* A simple constant. This is only allowed if the constant is intended to be used with preprocessing directives.

Example:

#define NUM\_RAILS 5

#if (NUM\_RAILS >= 4)

* A string literal.

Example:

#define VERSION\_DATE\_LONG "15.11.2017"

* A parenthesized expression which includes function-like macros (see **§ 2.5.2**).

Example:

#define NUM\_SLOTS 10[[35]](#footnote-35)

#define ELEMENTS\_IN\_SLOT\_ARRAY ((t\_Ui8)(NUM\_SLOTS + 1))

for ( idx = 0 ; idx < NUM\_SLOTS ; idx++ )

t\_Si16 s\_GearContactOffset[ELEMENTS\_IN\_SLOT\_ARRAY];

🡺 (*pre-processing*)

for ( idx = 0 ; idx < 10 ; idx++ )

t\_Si16 s\_GearContactOffset[((t\_Ui8)(10+1))];

* A type qualifier.

Example:

#define APPCONST const

extern APPCONST t\_Sig\_Ui8TA m\_Swc\_SIps1;

🡺 (*pre-processing*)

extern const t\_Sig\_Ui8TA m\_Swc\_SIps1;

* A storage class specifier.

Example:

#define DESCNET\_USDT\_STATIC static

DESCNET\_USDT\_STATIC void Swc\_DoSomething(void);

🡺 (*pre-processing*)

static void Swc\_DoSomething(void);

In particular, macros shall not be used to redefine statements and parts of statements or to redefine identifiers.

|  |
| --- |
| **Rule**: If a macro represents an expression, **usage of parenthesis is** **mandatory**. |

**Corollary #1**: In C, by default a constant has type ***signed integer***. Thus, in order to avoid Axivion violations when using a C-macro in expressions (mixing up signed and unsigned integers), a typecast is needed when defining it. However, putting a typecast before a constant transforms the constant into an expression. As a direct consequence, the expression must be enclosed in parentheses.

**Corollary #2**: In C, a negative number is seen as a unary negative operation on a positive constant and thus is an expression. Hence a negative number SHALL be put between parentheses.

Not putting parentheses around an expression can lead to unexpected behaviour as this simple mistake cannot be detected by the compiler due to the definitions of the ISO-C standard.

**Example #1**: (for clarity, typecasts have been omitted)

#define NUM\_SOMETHING 4

#define ELEMENTS\_IN\_SOMETHING NUM\_SOMETHING + 1

#define ARRAY\_SIZE 1 + 2 \* ELEMENTS\_IN\_SOMETHING

t\_Si16 some\_array[ARRAY\_SIZE];

After macro expansion, the code becomes:

t\_Si16 some\_array[1 + 2 \* 4 + 1];

Because the ‘\*’ operator has precedence over the ‘+’ operator, the above line is equivalent to:

t\_Si16 some\_array[1 + (2 \* 4) + 1];

And the result is an array with 10 elements. On the other hand, if the macros are declared as follows:

#define ELEMENTS\_IN\_SOMETHING (NUM\_SOMETHING + 1)

#define ARRAY\_SIZE (1 + 2 \* ELEMENTS\_IN\_SOMETHING)

Then the code becomes (after macro expansion):

t\_Si16 some\_array[(1 + (2 \* (4 + 1)))];

And the array has now 11 elements, which is the expected result.

In both cases, the code will compile just fine, but in the first case, because the parenthesis are missing, we end with an array that has the wrong size, thus leading to probable runtime errors when someone will try to access the 11th elements.

**Example #2**:

#define CONSTANT -50

And the following code:

SomeVariable = -CONSTANT ;

Then, depending on the compiler, SomeVariable will contain either **50** [-(-50)] or **49!** [--(50)]

**Note**: If your macro has the wrong format, a Misra-C 2012 20.4 violation will be raised.

Macros shall not be undefined (i.e. #undef shall not be used)[[36]](#footnote-36). The only exceptions to this rule are compiler switches and undefining the locator macro (LOC) for the generic locator mechanism.

The benefits of avoiding compiler specific #pragmas for memory location are weighted higher than the possible consequences of violating the rule. Note that this mechanism also implies additional limitations on the length of identifiers used for objects, which are subject to a non-standard memory location (see **“Method Description Engineering – Naming Convention in C”** document for more information).

## Function-like macros

Though macros can provide a speed advantage over functions, a function shall be used in preference to a function-like macro[[37]](#footnote-37), as it provides more robust mechanism (e.g. checking of parameter types or the problem of potential multiple evaluation of macro parameters).

If a function-like macro is to be used, then in its definition, not only the complete expression but also its parameters must be enclosed in parentheses[[38]](#footnote-38).

NOT ALLOWED: #define MIN(a, b) (a < b) ? a : b

CORRECT: #define MIN(a, b) (((a) < (b)) ? (a) : (b))

## Conditional compilation

Definition and undefinition of macros can be used for pre-compile time configuration of software. Such configuration may be required on two different levels, generic code archive (i.e. SW components) and project specific code archive (i.e. vehicle projects).

All macro identifiers used in preprocessor directives must be defined before use, except when used with the #if defined() directive[[39]](#footnote-39). Otherwise some preprocessors do not give any warning but assume the macro identifier to be zero.

The

#ifdef <CS>

And

#ifndef <CS>

Preprocessor directives shall not be used, as the different syntax is misleading.

*Please note that the braces around CS are used to improve readability in this document but of course must not be used in a real C program.*

## Generic code conditional compilation

Note that this is one of the exceptions where #undef may be used and that this exception is limited to the usage in the compiler switch files (e.g. ConfigCmn.h). The macro is then referred to as a compiler switch. Furthermore, there shall be at least one explicit definition or undefinition in all projects for any compiler switch that is used in the code.

For generic code variant handling, conditional compilation must be achieved by using the following preprocessor directives:

#if defined (<CS>)

Or

#if !defined (<CS>)

## Project specific code conditional compilation

For project specific code, conditional compilation must be achieved by applying a relational operator (like ==, >=, <=, etc.) to a macro constant for project specific variant handling, thus leading to the following preprocessor directive format:

#if (<variant\_constant> <rel\_op> <variant value>)

The variant\_constant shall be of format VARIANT\_<VARTEXT>, whereas the variant\_value shall be of format VAR\_<VARTEXT>\_<VARSPECIFIER>. Each variant\_constant available for a project shall be set in the configuration file for a project (e.g. config.xml) to one of the variant\_values defined in the related header file (e.g. variantdefs.h). Usage of these macro constants shall be limited to the project specific code archive.

**Example**:

#if (VARIANT\_TCUTYPE >= VAR\_TCUTYPE\_C0)

## Include statements

ISO-C defines two possible representations of an include statement[[40]](#footnote-40)

#include <filename>  
#include “filename”

Though the behaviour of the different representations is implementation-defined (i.e. left to the compiler builder, which has to document the behaviour) in the ISO-C standard, the rationales included in the ANSI-C standard and the K&R[[41]](#footnote-41) compendium give recommendations on the usage of the two different representations. The <> representation is meant to search some standard places (i.e. predefined by the compiler), whereas the “” representation searches the current directory first and then the same standard directories (which is one of the few definitions in the standard). These recommendations are widely accepted and lead to the following conclusions:

The <filename> representation shall be used for the standard libraries’ (i.e. delivered with the compiler) header inclusion, whereas the “filename” representation shall be used to include any other file[[42]](#footnote-42).

Files shall be included from their location in the archive by adding their absolute path to the compiler search path whenever possible, unless the file is autogenerated or is project or platform specific and has no unique filename or defined archive location. Such files shall be copied to the projects include directory and this copy shall be included.

Note that relative paths shall not be used as one shall neither use the “\” path delimiter (is the escape character in ISO-C[[43]](#footnote-43) and the behaviour is undefined if included in a include filename) nor the “/” path delimiter (as this may not be recognised as a path delimiter depending on the platform).

The standard limits unique mapping for included files to 6 letters (no digits), a period and a final single character. As the include syntax always depends on implementation-defined behaviour any compiler used shall be checked to ensure that it supports significance of all characters (including digits) in the filename.

## Pragmas

The ISO-C standard only defines, that a #pragma preprocessing directive will make the compiler behave in an implementation defined way and that unknown #pragmas shall be ignored by the compiler[[44]](#footnote-44).

As this definition is in conflict with source code portability #pragmas shall not be used. The only exception to this rule are #pragmas used for locating objects to different memory segments in the controller (i.e. constants to calibratable memory, variables to EEPROM or NVRAM, all functions of a process to specific memory sections).

To keep the compiler specific #pragmas for locator control away from the code a generic locator control mechanism has been defined. For details on this mechanism please refer to section 2.11.

## Functions

## Declaration and Definition

There shall be at most one function definition per c-file and functions must not be defined in header files[[45]](#footnote-45). When creating new functions, the template for the standard editor shall be used[[46]](#footnote-46).

All functions shall have prototype declarations and the prototype shall be visible at both the function definition and call[[47]](#footnote-47), and these declarations shall be at file scope[[48]](#footnote-48) (not at block scope, i.e. inside another functions body). Identifiers shall be given for all parameters in a function prototype[[49]](#footnote-49).

**Example**:

WRONG: void Scw\_CalcSomething(t\_Si16, t\_Si16);

CORRECT: void Scw\_CalcSomething(t\_Si16 param\_1, t\_Si16 param\_2);

Whenever a function is declared or defined, its return type shall be explicitly stated[[50]](#footnote-50). The value returned from a function is returned by value, i.e. it behaves just the same as if the result of the function was assigned to an object. Therefore, one can only return objects from a function that can be directly assigned, like single values, structures or pointers. Arrays or functions cannot be returned from a function, unless they are passed by reference, i.e. a pointer to them is returned[[51]](#footnote-51).

In this context it is crucial to be aware of the binding rules defined in the ISO-C standard for declaring the type of a function.

WRONG: t\_Si16 \*Swc\_CalcSomething(void);

CORRECT: (t\_Si16 \*) Swc\_CalcSomething(void);

Declares a function returning a pointer to an SI16,

While:

t\_Si16 (\*Ptr\_CalcSomething(void));

Declares a pointer to a function returning an SI16.

Therefore, parentheses shall be used to emphasize the binding whenever declaring a function returning a pointer type or a function pointer.

For each function parameter the identifier[[52]](#footnote-52) and type[[53]](#footnote-53) given in the declaration and definition shall be identical, and the return types shall also be identical. For a type to be identical the typedef name and type qualifiers must match. Functions shall be defined with fixed numbers of arguments[[54]](#footnote-54) and functions with no parameters shall be declared with parameter type void[[55]](#footnote-55).

## Parameters and return values

The C language provides different means of passing values to and returning modified values from a function.

* global variables
* function parameters (pass by value, pass by reference)
* return value

The simplest but at the same time least flexible way is using global variables. Using global variable to pass data to a function hides these inputs from the programmer and at the same time binds the function to the context of these global variables making it almost impossible to reuse it in a different context. The same arguments apply for using global variables to return results from function calls, as the side-effects caused by a function are hidden inside the function definition. Therefore, it is recommended to avoid usage of global variables.

When new or modified values are to be returned from a function using the return value of that function is recommended for that purpose. As the rules of § **2.6.3** apply at the same time this makes it very easy to see which value is returned inside the function body.

Function parameters are recommended to be used for passing data to a function. The parameters declared in a function’s parameter list can be used inside the function as if they were local automatics, despite the fact, that they are initialized to the value passed when calling the function. This includes write access to the parameter altering its value at some point in the function’s code. As this can be confusing, write access to parameters shall be avoided and local variables shall be used for this purpose instead. It is recommended to declare all parameters for which write access is not assumed as const to allow the compiler to check for unintended write access

**Example**:

void Swc\_CalcSomething( const t\_Si16 in\_param)

Though C has no direct mechanism to distinguish between input parameters and input/output parameters this can be implicitly handled by the type declared for each parameter. If passed by value (only possible for single values and structures) the parameter is an input parameter, i.e. all changes done to the parameter inside the function are not visible to the outside, as the value has been copied to the local automatic representing the parameter in the function. If the changes done to a parameter shall affect the object passed to the function it must be passed by reference, i.e. as a pointer. Array type objects can only be passed this way.

**Example**:

void Swc\_CalcSomething( t\_Si16 \*inout\_param\_ptr);

A pointer parameter in a function prototype should be declared as “pointer to const” if the pointer is not used to modify the addressed object (i.e. the parameter addressed is only read and not written inside the function)[[56]](#footnote-56).

**Example**:

void Swc\_CalcSomething( t\_Si16 \*inout\_param\_ptr, const t\_Si16 \*in\_param\_ptr)

{

\*inout\_param\_ptr = \*inout\_param\_ptr + \*in\_param\_ptr;

}

Note that both mechanisms can be applied at once, i.e. a parameter passed as pointer can be declared to be a “constant pointer to constant value”, which means, that neither the pointer will be changed by writing to the parameter nor the object referenced will be changed.

**Example**:

void Swc\_CalcSomething( const t\_Si16 \* const in\_param\_ptr)

Complex evaluations to determine the parameters inside a function call shall be avoided.

**Rationale**: The result of an expression shall be the same under any evaluation order that the standard permits[[57]](#footnote-57). This is an issue if evaluation causing side effects (assignments, function calls) are nested into a function call, as the ISO-C:90 standard leaves the order in which the function parameters are evaluated to the compiler. As static checking would require detailed analysis of the side-effects caused by function calls it is hard to be checked by a static code checker. Therefore, strictly avoiding complex evaluations in function calls is the simplest way of reducing the risk of critical failures.

In case function parameters are not used in all variants of a function (e.g. if compiler switches are used) the compiler will create warnings for unused parameters. For explicitly not used parameters, the warning can be deactivated by an explicit cast to void. A comment shall be added to clarify the intention.

**Example**:

(void) diag\_req\_ips;

(void) \*diag\_error\_ptr;

Explicitly not evaluated return values of a called function may also be casted to void in order to avoid compiler warnings.

**Example**:

(void) DiagRoutGenBothCl\_Ca(method, TEST\_REFILL\_CA, ...);

## Termination

A function shall have a single point of exit at the end of the function (i.e. there shall be no or a single return statement at the end of a function)[[58]](#footnote-58) and the return statement of a function with none void return type shall have an expression[[59]](#footnote-59).

## Function calls

The number of arguments passed to a function shall match the number of parameters in the prototype[[60]](#footnote-60) and empty parentheses shall be used if the parameter list is empty[[61]](#footnote-61).

The value returned by a function shall be tested (e.g. assigned to a variable or used in an if-clause)[[62]](#footnote-62).

Recursive function calls, i.e. a function calling itself either directly or indirectly, shall not be used[[63]](#footnote-63), unless the call depth is known and static, i.e. cannot change during execution. Rationale: The worst-case stack usage and runtime requirements of recursive codes are hard to predict, or control and consequences can be serious.

## Function pointers

The C language features a very powerful mechanism for generating generic code by using so called function pointers, which means that when calling a function, it can be identified by its address instead of its function name. Though being a very powerful mechanism, the concept of function pointers is complex and hence critical at the same time. Therefore, function pointers shall not be used unless accepted by the SW architect in the design review and reviewed by two persons familiar with the concept of function pointers in the implementation review.

When using function pointers, a function type or function pointer type shall be declared, and this type shall then be used to declare the objects for storing the pointers to functions of that type.

**Example**:

Declaring a pointer type to a function with no return value (void) and a single constant input parameter of type t\_Si16:

typedef void (\*t\_Func\_Ptr)( const t\_Si16 in\_param)

Defining different functions compatible to that type:

void Swc\_FuncSomething ( const t\_Si16 in\_param) {…}

void Scw\_FuncSomethingElse ( const t\_Si16 in\_param) {…}

Using the function pointer type for defining a constant array of two function pointers pointing to that functions:

const t\_Func\_Ptr c\_FuncPointersTable[2] = {

&Swc\_FuncSomething,  
 &Swc\_FuncSomethingElse

};

And finally calling a function referenced by one of the elements in the array

(\*(c\_FuncPointersTable[0]))(<some 16-bit signed value>);

Which is equivalent to calling the function Func1 directly

Swc\_FuncSomething (<some 16-bit signed value>);

A pointer to a function shall never be converted to a different type, as the behaviour resulting from doing so is undefined[[64]](#footnote-64).

## Passing an array as parameter to a function

Arrays as parameters in a function are treated in the same way as pointers. This has the consequence that the function receives a pointer to the first element of the array. Furthermore, the array is not passed as a copy, but can be modified in the function as long as the parameter is not specified as const.

There are two equivalent ways of declaring a function taking an array as parameter. Both declarations allow the modification of the array content from inside the function. The preferred variant is to use my\_array[] as this clearly indicates that the parameter is an array.

**PREFERED**:

t\_Ui8 MyArrayFuncArrayParam(t\_Ui8 my\_array[], t\_Ui8 my\_array\_size)

{

t\_Ui8 sum = 0;

for( t\_Ui8 i = 0; i < my\_array\_size; i++ )

{

sum += my\_array[i];

}

my\_array[0] = sum / 2; /\* Manipulation of array from inside \*/

/\* of the function is possible \*/

return sum;

}

**OPTIONAL**:

t\_Ui8 MyArrayFuncPtrParam(t\_Ui8 \* my\_array, t\_Ui8 my\_array\_size)

There are two equivalent ways of passing an array during a function call. Inside the function the element operator can be used in both cases.

t\_Ui8 array[] = {1, 2, 3, 4};

result = MyArrayFuncArrayParam(array, sizeof(array));

result = MyArrayFuncArrayParam(&array[0], sizeof(array));

result = MyArrayFuncPtrParam(array, sizeof(array));

result = MyArrayFuncPtrParam(&array[0], sizeof(array));

As the array is passed as a pointer, the content of the pointer / array can be modified. See also **§ 2.4.4**. To ensure that the passed array content cannot be modified within the function it is passed to, the array parameter of the function must be declared as const.

**PREFERED**:

t\_Ui8 MyArrayFuncConstArray(const t\_Ui8 my\_array[], t\_Ui8 my\_array\_size)

**OPTIONAL**:

t\_Ui8 MyArrayFuncConstPtr(const t\_Ui8 \* my\_array, t\_Ui8 my\_array\_size)

## Operators

Limited dependence or no dependence at all shall be placed on C’s operator precedence rules in expressions[[65]](#footnote-65). Though exactly specified in ISO-C90 operator precedence is quite complicated and may easily lead to errors. Therefore, operator precedence shall be overwritten or emphasized using parentheses. Beyond that operators that are associative in algebra are not necessarily associative in ISO-C.

The following rules apply:

1. Assignments must not be nested.

NOT ALLOWED: a = b = c + d

1. The complete right-hand side of an assignment does not need to, but can be parenthesized.
2. Unary operators[[66]](#footnote-66) do not need to be parenthesized.

ALLOWED: if (

(Something\_B == TRUE)

&& !(a > b) /\*unary NOT is OK\*/

)

1. Any binary or ternary operator does need to be parenthesised, unless all operators are the same.
2. Even if all operators are the same, it may be reasonable to parenthesize them (e.g. to avoid an overflow by first subtracting a value before adding another one)
3. The unary minus operator shall not be applied to an expression whose underlying type is unsigned[[67]](#footnote-67).
4. The comma operator shall not be used[[68]](#footnote-68).

NOT ALLOWED:

x = 2, x+1;

for ( i = 1, x = 10; i < IDX\_MAX ; i++, x--) …

1. The ‘?’ ternary operator shall not be used in C code. It may be allowed in some macros for efficiency or legacy reasons (**MIN** and **MAX** for example). However, before introducing a new macro with that operator, please contact a software architect.

NOT ALLOWED:

t\_Si32 n\_ips\_current;

n\_ips\_current = ((tgt\_ips != IPS\_UNKNOWN) ? \

Swc\_NIpsAbsAt[tgt\_ips].New) : (0L));

CORRECT WAY:

#define N\_IPS\_UNKNOWN ((t\_Si32) 0)

if ( tgt\_ips != IPS\_UNKNOWN ) {

n\_ips\_current = NIpsAbsAt\_Sig[tgt\_ips].New ;

} else {

n\_ips\_current = N\_IPS\_UNKNOWN ;

}

1. The value of an expression shall be the same under any order of evaluation that the standard permits[[69]](#footnote-69). This cannot be achieved by using parenthesis, as it is not a matter of operator precedence. To avoid possible problems, it is recommended to avoid nesting operators or functions causing side effects into an expression. Especially the increment (++) and decrement operators (--) can be dangerous, as the side effects they cause are not necessarily obvious.

BAD EXAMPLE:

#define MIN(a, b) (((a) < (b)) ? (a) : (b))

i = 1;

res = MIN(i++, 5);

In this example res is 3 and not 2 as one might expect, because i++ is evaluated twice.

## Control flow

The goto and continue statements shall not be used[[70]](#footnote-70). Furthermore, the break statement shall only be used to terminate cases of a switch expression. For Premature termination of a for-loop please refer to **§ 2.8.3**.

## if

Any if constructs (this includes simple if and if-elsif constructs) shall have a final else-clause[[71]](#footnote-71). Note that adherence to this rule requires to add at least a comment to any empty else-clause in the Innovator editor, as it will otherwise not generate the else-clause in the source-file while displaying it on the screen. Rationale: Not adhering to this rule may cause different compilation results if the else-clause is added later, even if it is empty, e.g. only containing comments or code deactivated by compiler switches. Therefore hex-identity expected might not be maintained by such changes. This requirement is stricter than the one in the MISRA-C standards.

The conditions of an if often contain complex expressions using the logical operators (&&, ||, !). All operators in the conditions shall be effectively Boolean[[72]](#footnote-72) (e.g. comparisons) and only the operators mentioned before shall be used on them. The operands to the logical operators shall be primary expressions[[73]](#footnote-73) (Essentially either a single identifier or a constant or a parenthesized expression) and shall not contain side effects[[74]](#footnote-74) or assignments[[75]](#footnote-75) (e.g. assignments or function calls causing side effects, i.e. changing the value of a variable).

Test of a value against zero shall be made explicit[[76]](#footnote-76) and Boolean variables shall explicitly be tested against TRUE or FALSE.

Note that the rules for not relying on operator precedence from **§ 2.7** also apply to the logical operators.

To improve readability and establish a common code layout the following additional rules apply.

* Each simple boolean expression shall be placed on a line by itself
* Logical operators shall precede the right-hand operand
* If a variable is tested against a constant the variable shall be the left-hand operand (this is not defensive as it would allow for erroneously using an assignment instead of a test for equality, but such errors are easily avoided by testing for forbidden side effects in the conditions)
* Indentation shall be used to emphasize the hierarchy in a complex condition

NOT ALLOWED:

if ( a > b && !5 <= c || TRUE == b\_finished)

DESIRED:

if (

(a > b)

&& (

!(c >= 5)

|| (b\_finished == TRUE)

)

)

## switch

Any switch-statement shall contain a final default clause[[77]](#footnote-77), shall have at least two case clauses[[78]](#footnote-78) and any non-empty case clause shall be terminated by an unconditional break-statement[[79]](#footnote-79).

## for-loop

The three expressions of a for statement shall be concerned only with loop control[[80]](#footnote-80), numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop[[81]](#footnote-81) (i.e. to prematurely terminate the loop) and a for-loop shall not be prematurely terminated using a break-statement (use a do-while loop or an additional flag for dual outcome loops instead).

## Comments

All comments shall be written in English.

The rule is: better too many comments than too few. Comments have to be relevant. If the code is modified, the appropriate comments must also be modified. For this reason, the comment should be written next to the code. In the head of each routine there will be a description of its function: Its purpose, what process/SW component it belongs to, what kinds of arguments are used, etc. The way how this functionality will be turned into a code should be commented directly in the code.

As a standard, the Innovator-Comment for a block shall be used (in the Nassi-Shneiderman-Editor invoked with “F2”). This comment is printed in a different font / colour at the top of the block.

Standard (Single graphical block with comment on top):

|  |
| --- |
| **Local copy of clutch torque:**  t\_Si16 trq\_cl; |
| **Input shaft index:**  t\_Si16 idx\_shaft; |

The only exception to this rule is commenting the single conditions of a complex if-clause and commenting the single members of structures, enumerations or a set of macros belonging together (e.g. state numbers).

Allowed (e.g. for structure members):

|  |
| --- |
| **Structure for signals:**  typedef strcut  { t\_Si16 Old; /\* Signal of last reading \*/  t\_Si16 New; /\* Current valid Signal \*/  t\_Si16 Status; /\* Status (valid, too high… \*/  } typSignal; |

Only /\* … \*/ style comments shall be used in the code[[82]](#footnote-82) and comments must not be nested (i.e. the character sequence /\* or // must not be used inside a comment[[83]](#footnote-83)).

Though most compilers support both non-standard features (//-style comments and nested comments) as an extension to the ISO-C standard this is not guaranteed for all compilers and hence limits portability.

Furthermore, it is possi­ble that other compilers interpret the same code in a different way depending on how they interpret and prioritize nested comment delimiters.

|  |
| --- |
| /\* Example of very bad comment combination: \*/ int test( void )  {  return 4 //\* divisor \*/ 2  ;  }  /\* The above will return the value 2 on a compiler that does not \*/  /\* allow C++ comments but will return 4 if it allows C++ comments. \*/ |

Source code shall not be “commented out”[[84]](#footnote-84).Unused source code shall be deleted instead. To allow for easy static checking characters typical for C-sources like a semicolon at the end of lines or curly braces ({,}) shall not be used inside comments.

Comments do not sum up the program code in English language but describe on a higher abstraction level which objective or which functionality should be realized with this code (pseudo code). The comment should describe “why” something is done and the code will show “how”. It is recommended to comment on “how” the functionality is implemented if complex or non-standard mechanisms (e.g. function pointers) are used to avoid misunderstandings by developers not used to the construct.

For a global variable or constant definition, the following has to be specified as comment:

* Physical unit (especially for signals, e.g.: deci-Nm)
* Range of software values
* Resolution
* Replacement value in case of signal failure (if there is one)
* Update rate
* Description

It is recommended to add the same information at definition of local variables.

**Example**:

|  |
| --- |
| **Unit: [Nm]**  **Range: 0...5000**  **Resolution: 0.1**  **Replacement: N/A**  **Update rate: 10ms**  **Description: This is an array for estimated actual clutch torque, with input shaft number as array index. The clutch torque is calculated based upon estimated friction coefficient, touch point and expected torque-position curve.** |

## C – Code editor

The mainline software consists of many files; most of them are coded in the C-language. In order to have a common “look” of the code (indentation, spacing, line length, etc.), the following rules apply for code (.c) and header (.h) files:

The standard editor is the Innovator-Nassi-Shneiderman Editor (InoSp.exe). This editor displays code and comments with graphical block elements, so the control structures become more visible. As a general rule, those graphical elements must be used.

The only exception to this rule is made for autogenerated code or header files with limited lifetime, i.e. files generated by a tool for compilation that are not a permanent part of the SW archive (e.g. config.h or ICE interface headers), third party files or files delivered to the customer.

Where available the templates shall be used for creating new files. The templates can be found in the Templates\_CM subfolder of the process area this document is related to.

The Innovator-Editor saves the code as a normal text file, where the graphical information is coded into C-comments (e.g.: /\* Ino.10180.20180.000C0 \*/). The file can be looked at with a plain (“flat”) text editor, but it must not be modified with a flat editor, because otherwise the consistency of that graphical information might be destroyed. In that case, the file is corrupted and cannot be opened with the Innovator editor anymore!

## Style guide for the Nassi-Shneiderman-Editor (InoSp.exe)

## Configuration

There are several settings in the configuration of the Innovator Nassi-Shneiderman-Editor that influence the formatting of the resulting output file. For comparability of the code files before and after a change, it is mandatory that the following settings are identical for each user:

Line length 80  
 Element content source and description  
 Blanks for Tabulator 4  
 . . . .

**Important**: The above settings are just an example. The settings may differ with each version of the Innovator, and therefore are not fixed in this work instruction. The settings relevant for the current Innovator version will be specified and published in a reasonable manner by the Administrators of Function and Software Development @ BU eMobility for the Innovator in Bühl.

There are additional styles and options that can be imported via the file “inotxt.ini”. Techni­cally, this file can set forward the way how certain control structures are formatted in the result file. Such elements are not allowed for the PSG software! It is allowed to import the style file “inotxt.ini”, but it must not change any formatting. It may e.g. set the default graphi­cal look of newly added elements.

The settings of the display colours and fonts etc. may be set by each user according to his/her likes.

## Zoom – Status

The Innovator-Editor allows grouping one or several graphical blocks, and “collapsing” this “super block” into just a small graphical element. By this, complex code can be “hidden” and the user just sees a comment that summarizes the code behind it. This can help very much to keep an overview of the software; on the other hand, this can lead the programmer to put lots of code into a single routine or file.

The innovator saves the zoom-status (collapsed or expanded) of every grouping block to the file via C-comments. If no code change has been done, but the blocks have changed their zoom-status, those changes will be visible in a file compare. In order to keep the number of changes due to innovator zoom-status at a minimum, the user shall collapse all blocks inside a C-routine.

The file will then typically look like this:

+ **Description**

+ **Includes**

t\_Si16 SomeFunction( void )

+ **Local Variables**

+ **Module Code**

As can be seen, also the “Description” and “Includes” blocks etc. shall be collapsed.

## Structural elements

The Nassi-Shneiderman offers graphical elements for almost all control structures of the C language, like “if-then-else”, “switch-case-default”, “for(;;)”, “while()”, etc. These graphical elements must be used, because they improve the readability of the code and visualize the control flow. The only exception to this rule is conditional compilation (compiler switches) inside a function parameter list or an if-condition, though it is recommended to duplicate the function or if-block in this case.

## Locator control

After compilation the linker allocates all objects and code to physical memory in the controller for generating an executable hexfile. The memory classes available are controller and project specific and include ROM (for code and constants), RAM (initialized / uninitialized), EEPROM, NVRAM (reset safe). These classes are often split into sub-sections for projects (e.g. to have separate sections for different SW components or the parties involved in a project).

Memory location is often controlled by assigning some predefined memory classes to functions and objects and redirecting these classes to project defined memory sections. The compiler/linker specific mechanism is part of the compiler/linker documentation (i.e. is implementation defined). Pragma directives are commonly used for control (see **§ 2.5.5**).

To keep compiler specific #pragmas away from the code in order to simplify the code portability to different platforms a generic locator control mechanism has been defined.

This and no other mechanism shall be used to control memory location of functions and objects. The mechanism must be implemented for each single constant and may be used for any other object or function. The mechanism uses a platform specific include file (locator.h) with all the #pragmas or other control syntax required. This file evaluates the locator macro LOC and behaves accordingly. The file also resets the locator control to the default classes for the project if the locator control macro is undefined. The generic locator control macro (LOC) is redefined to an object/function specific locator control macro, the latter being defined by a central and project specific locator control header file. For details on the naming convention please refer to the **“Method Description Engineering – Naming Convention in C”** document.

**Example**:

#define LOC L\_c\_Swc\_SomeCalibrationConstant  
 #include “locator.h”  
#undef LOC

const t\_Si16 c\_Swc\_ SomeCalibrationConstant = 250;

#include “locator.h”

The generic mechanism will be provided by the common part of the make-environment for any platform supported.

## File names

The names of c-files and process files as well as the file and include structure are described in **S0002\_FileAndIncludeStructure\_Arc.docx**.

Before introducing a new process file and deciding a process number (e.g. p461\_gb.c) please contact a software architect.

## Known Compiler Problems and Containments

The known compiler problems by target processor, compiler vendor and version are documented in the S0020 document in the 2019 docu collection and hence is available in each sandbox, see **S0020\_KnownCompilerProblemsAndContainments\_Env.docx** for more information.

It shall be used as a reference for each project and the developers of generic code to determine which measures must be taken to avoid these problems.

# Templates, Checklists, and Tools

| **No** | **Date (Version)** | **Title / Description** | **Author** | **Path** |
| --- | --- | --- | --- | --- |
| 1 | 19.11.2008 (1.1) | C-file template for functions | Stehle | [tmpl\_cxxxyy\_PsgFcnName\_swc.c](file:///\\emea.luk.com\Buehl\DATA\ALGBHL-L\Projects\LXES_Projects\CMMI_LuK\Process_Library\Released_CM\Engineering\Templates_CM\tmpl_cxxxyy_PsgFcnName_swc.c) |
|  |  |  |  |  |
|  |  |  |  |  |

# References / Further applicable Documents

| **No** | **Date (Version)** | **Title / Description** | **Author** | **Path** |
| --- | --- | --- | --- | --- |
| 1 | 1990 | ISOIEC 9899:1990 / Programming Language – C (with technical corrigendum 1 – 1995) | ISO | [ISO IEC 9899:1990](file:///\\emea.luk.com\Buehl\DATA\ALGBHL-L\Projects\LXES_Public\Standards\Programming_Languages\ISO-C\ISO_9899_1990\ISO_IEC_9899_1990_Scanned_whole_doc.pdf) |
| 2 | 1999 | ISO/IEC 9899:1999, Programming languages — C, International Organization for Standardization, 1999. | ISO | [ISO IEC 9899:1999](file:///\\emea.luk.com\Buehl\DATA\ALGBHL-L\projects\LXES_Public\Standards\Programming_Languages\ISO-C\ISO_9899_1999\ISO_IEC_9899_1900_2000.pdf) |
| 3 | 1998 | MISRA-C:1998 / Guidelines for The Use of The C Language In Vehicle Based Software | MISRA | [MISRA-C:1998](file:///\\emea.luk.com\Buehl\DATA\ALGBHL-L\Projects\LXES_Public\Standards\SW_Quality\MISRA\misra-1998.pdf) |
| 4 | 2004 | MISRA-C:2004 / Guidelines for the use of the C language in critical systems | MISRA | [MISRA-C:2004](file:///\\emea.luk.com\Buehl\DATA\ALGBHL-L\Projects\LXES_Public\Standards\SW_Quality\MISRA\misra-2004.pdf) |
| 5 | 2012 | MISRA-C:2012 / Guidelines for the use of the C language in critical systems | MISRA | [MISRA-C:2012](file:///\\emea.luk.com\Buehl\DATA\ALGBHL-L\Projects\LXES_Public\Standards\SW_Quality\MISRA\MISRA%20C2012%20Guidelines%20for%20the%20use%20of%20the%20C%20language%20in%20critical%20systems.pdf) |
| 6 | 2016 **(\*)** | Method Description Engineering – Programming in C | Björn Stehle,  Gerald Pfeiffer,  F. Menneteau | [md\_C\_Programming\_Guideline.doc](integrity://integrity:7001/si/viewproject?project=/eMob/PMT/Instructions/RnD_SW/Process_Library/Engineering/Method/project.pj&selection=%22md_C_Programming_Guideline.doc%22) |
| 7 | 2008-03 (1.1) | md\_MISRA\_Check | Bechmann |  |
| 8 | 2020 | Naming Convention Rules in ASW 2019 | Francois Menneteau | [S0029\_NamingConventionRules.docx](integrity://integrity:7001/si/viewproject?project=/eMob/PL/Software/Docu_Collection_ASW_2019/01_sw_components/arc/03_docu/project.pj&selection=S0029_NamingConventionRules.docx) |
| 9 | 2019 | Configuration Management ASW\_2019 Mainline | Björn Stehle,  Gerald Pfeiffer,  F. Menneteau | [S0002\_FileAndIncludeStructure\_Arc.docx](integrity://integrity:7001/si/viewproject?project=/eMob/PL/Software/Docu_Collection_ASW_2019/01_sw_components/arc/03_docu/project.pj&selection=S0002_FileAndIncludeStructure_Arc.docx) |
| 10 | 2019 | Known Compiler Problems and Containments | Gerald Pfeiffer,  F. Menneteau | [S0020\_KnownCompilerProblemsAndContainments\_Env.docx](integrity://integrity:7001/si/viewproject?project=/eMob/PL/Software/Docu_Collection_ASW_2019/05_env/03_docu/project.pj&selection=S0020_KnownCompilerProblemsAndContainments_Env.docx) |

**(\*) Note: Valid version of documents is the latest revision with state "Released" in ILM**

# Change history

| **Author** | **Date** | **ILM Version** | **Change** |
| --- | --- | --- | --- |
| F. Menneteau | xx.03.2020 | 1.1 | Creation (based on revision 1.12 of [6]) |
| F. Menneteau | 19.01.2022 | 1.6 | § 2.13 “Usage of Veh vs. Cif vs. SWC” has been removed. This is now described in the SAD PF as a Cross Cutting Concept. |
|  |  |  |  |
|  |  |  |  |

1. MISRA-C:2012 Directive 4.6 [↑](#footnote-ref-1)
2. ISO-C:99 section 6.7.8 [↑](#footnote-ref-2)
3. MISRA-C:2012 rule 9.1 [↑](#footnote-ref-3)
4. MISRA-C:2012 rule 18.6 [↑](#footnote-ref-4)
5. for details see ISO-C:99 section 6.7.8 [↑](#footnote-ref-5)
6. MISRA-C:2012 rule 8.9 [↑](#footnote-ref-6)
7. ISO-C99 chapter 6.2.4 defines any object “whose identifier is declared with no linkage and without the storage-class specifier static” to be of “automatic storage duration”. [↑](#footnote-ref-7)
8. MISRA-C:2012 rule 8.7 [↑](#footnote-ref-8)
9. MISRA-C:1998 rule 28 [↑](#footnote-ref-9)
10. Swc stands for “*Software Component*”, see **S0029\_NamingConventionRules.docx** for more information about naming entities. [↑](#footnote-ref-10)
11. MISRA-C:2012 rule 8.5 [↑](#footnote-ref-11)
12. MISRA-C:1998 rule 28 [↑](#footnote-ref-12)
13. MISRA-C:2012 rule 5.7 [↑](#footnote-ref-13)
14. ISO-C:99 section 5.2.4.1 Translation limits [↑](#footnote-ref-14)
15. On the big-endian architecture, the memory location of FirstMember and ThirstMember would differ but the total size in memory would be the same. [↑](#footnote-ref-15)
16. MISRA-C:2012 rule 6.1 and 6.2 [↑](#footnote-ref-16)
17. MISRA-C:2012 Directive 1.1 [↑](#footnote-ref-17)
18. ISO-C:90 sections 6.3.8, 6.3.9, 6.3.13, 6.3.14 [↑](#footnote-ref-18)
19. Be aware that although the result of a Boolean expression is defined to be 1 (TRUE) or 0 (FALSE) in ISO-C:90 the result is an int (signed) while typUBS is of type unsigned int. This leads to an implicit signed to unsigned conversion. In Misra-C 2012 10.1, as long as both types are declared as of boolean type, no violation will be raised. [↑](#footnote-ref-19)
20. For more information about the HIS.VOCF metric and how a Boolean expression can help reduce metrics complexity, see **§ 5** in **01\_swc\_components/arc/03\_docu/S0013\_Axivion\_Software\_Metrics.docx**. [↑](#footnote-ref-20)
21. MISRA-C:2012 rule 19.2 [↑](#footnote-ref-21)
22. MISRA-C:2012 rule 8.11 [↑](#footnote-ref-22)
23. ISO-C:99 section 6.7.2.2. [↑](#footnote-ref-23)
24. MISRA-C:2012 rule 9.2 and rule 9.3 [↑](#footnote-ref-24)
25. MISRA-C:2012 rule 4.1 and rule 7.1 [↑](#footnote-ref-25)
26. ISO-C:99 section 6.5.3.2 Address and indirection operators [↑](#footnote-ref-26)
27. MISRA-C:2012 rule 11.1, rule 11.2, rule 11.4 and rule 11.6 [↑](#footnote-ref-27)
28. ISO-C:99 section 6.7 Declaration [↑](#footnote-ref-28)
29. See also **§ 2.3.1** [↑](#footnote-ref-29)
30. MISRA-C:2012 rule 11.8 [↑](#footnote-ref-30)
31. MISRA-C:2012 rule 18.5 [↑](#footnote-ref-31)
32. MISRA-C:2012 rule 18.1, 18.2 and 18.4 [↑](#footnote-ref-32)
33. MISRA-C:2012 rule 11.3 [↑](#footnote-ref-33)
34. MISRA-C:2012 rule 20.4 [↑](#footnote-ref-34)
35. See also section **Error! Reference source not found.** for restrictions. [↑](#footnote-ref-35)
36. MISRA-C:2012 rule 20.5 [↑](#footnote-ref-36)
37. MISRA-C:2012 Directive 4.9 [↑](#footnote-ref-37)
38. MISRA-C:2012 rule 20.7 [↑](#footnote-ref-38)
39. MISRA-C:2012 rule 20.9 [↑](#footnote-ref-39)
40. ISO-C:990 section 6.10.2 Source file inclusion [↑](#footnote-ref-40)
41. Brian Kernighan, Dennis Ritchie: "The C Programming Language" [↑](#footnote-ref-41)
42. For details on file and include structure see S0005\_PSG\_File\_and\_Include\_Structure.doc [↑](#footnote-ref-42)
43. ISO-C:99 section 5.2.1 Character sets [↑](#footnote-ref-43)
44. ISO-C:99 section 6.10.6 Pragma directive [↑](#footnote-ref-44)
45. This restriction no longer exist in MISRA-C:2012 [↑](#footnote-ref-45)
46. See **§ 2.1** [↑](#footnote-ref-46)
47. MISRA-C:2012 rule 8.2, rule 8.4 and rule 17.3 [↑](#footnote-ref-47)
48. This restriction no longer exist in MISRA-C:2012 [↑](#footnote-ref-48)
49. MISRA-C:2012 rule 8.2 [↑](#footnote-ref-49)
50. MISRA-C:2012 rule 8.1 [↑](#footnote-ref-50)
51. ISO-C:99 section 6.7.5.3 Function declarators [↑](#footnote-ref-51)
52. MISRA-C:2012 rule 8.3 [↑](#footnote-ref-52)
53. MISRA-C:2012 rule 8.3 [↑](#footnote-ref-53)
54. MISRA-C:2012 rule 17.1 [↑](#footnote-ref-54)
55. MISRA-C:2012 rule 8.2 [↑](#footnote-ref-55)
56. MISRA-C:2012 rule 8.3 [↑](#footnote-ref-56)
57. MISRA-C:2012 rule 12.1 [↑](#footnote-ref-57)
58. MISRA-C:2012 rule 15.5 / IEC61508 Part 3 Table B.9 [↑](#footnote-ref-58)
59. MISRA-C:2012 rule 17.4 [↑](#footnote-ref-59)
60. MISRA-C:2012 rule 8.2 and rule 17.3 [↑](#footnote-ref-60)
61. MISRA-C:2012 rule 10.1 to 10.4 [↑](#footnote-ref-61)
62. MISRA-C:2004 Directive 4.7 [↑](#footnote-ref-62)
63. MISRA-C:2012 rule 17.2 [↑](#footnote-ref-63)
64. MISRA-C:2012 rule 11.1 [↑](#footnote-ref-64)
65. MISRA-C:2012 rule 13.1 [↑](#footnote-ref-65)
66. Operators working on a single operand. ISO-C:99 section 6.5.3 Unary operators defines 6 such operators: address of &, indirection \*, unary +, unary -, bitwise complement ~, logical NOT! [↑](#footnote-ref-66)
67. MISRA-C:2012 rule 10.1 [↑](#footnote-ref-67)
68. MISRA-C:2012 rule 12.3 [↑](#footnote-ref-68)
69. MISRA-C:1998 rule 47 / MISRA-C:2004 rule 12.1 [↑](#footnote-ref-69)
70. MISRA-C:2012 rule 15.1, 15.2 and 15.3 [↑](#footnote-ref-70)
71. MISRA-C:2012 rule 15.7 [↑](#footnote-ref-71)
72. MISRA-C:2012 rule 10.1 [↑](#footnote-ref-72)
73. MISRA-C:2012 rule 12.1 [↑](#footnote-ref-73)
74. MISRA-C:2012 rule 13.5 [↑](#footnote-ref-74)
75. MISRA-C:2012 rule 13.4 [↑](#footnote-ref-75)
76. MISRA-C:2012 rule 14.4 [↑](#footnote-ref-76)
77. MISRA-C:2012 rule 16.4 and 16.5 [↑](#footnote-ref-77)
78. MISRA-C:2012 rule 16.7 and 16.6 [↑](#footnote-ref-78)
79. MISRA-C:2012 rule 16.3 [↑](#footnote-ref-79)
80. MISRA-C:2012 rule 14.2 [↑](#footnote-ref-80)
81. MISRA-C:2012 rule 14.2 [↑](#footnote-ref-81)
82. MISRA-C:2012 rule 1.2 [↑](#footnote-ref-82)
83. MISRA-C:2012 rule 3.1 [↑](#footnote-ref-83)
84. MISRA-C:2012 Directive 4.4 [↑](#footnote-ref-84)