C Style Guide

**Change History**

| **Version** | **Date** | **Change Description / Reason** | **Author** |
| --- | --- | --- | --- |
| 1.0 | 2017-12-08 | Initial version  Delete chapter 2.9. Template for the Source Files  Add chapter 9. Required Coding Rules  Add Documentation rules (Ch 6.1 / 6.2 / 6.3/ 6.4) | Hae-Min, Woo |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[1. Coding guidelines constraints 5](#_Toc500344524)

[2. C Style Guide 5](#_Toc500344525)

[2.1 Upper and Lower Case 5](#_Toc500344526)

[2.2 Encapsulation of SW-Modules and SW-Components 5](#_Toc500344527)

[2.3 Minimal Function Amount 5](#_Toc500344528)

[2.4 Global Variables 5](#_Toc500344529)

[2.5 Missing Prototypes 6](#_Toc500344530)

[2.6 Constants 6](#_Toc500344531)

[2.6.1 Realization of Constants with Enumerations 6](#_Toc500344532)

[2.6.2 Comparison of Constants 6](#_Toc500344533)

[2.7 Defines 6](#_Toc500344534)

[2.8 Macros 6](#_Toc500344535)

[3. Filenames 7](#_Toc500344536)

[4. Header File 7](#_Toc500344537)

[4.1 Public / Private / Config Header 7](#_Toc500344538)

[4.2 Declarations and Definitions 8](#_Toc500344539)

[4.3 Including Header Files 8](#_Toc500344540)

[4.4 Including Header Files into C Files 8](#_Toc500344541)

[5. Name Conventions 8](#_Toc500344542)

[5.1 Variables 9](#_Toc500344543)

[5.1.1 Representation in Simplified Backus-Naur-Form (BNF) 10](#_Toc500344544)

[5.2 Name Conventions for Fields in struct and union 10](#_Toc500344545)

[5.3 Name Convention for typedef 11](#_Toc500344546)

[5.4 Name Convention for Constants 11](#_Toc500344547)

[5.5 Name Convention for Units (C Functions) 11](#_Toc500344548)

[5.6 C Module 11](#_Toc500344549)

[5.7 Header File 11](#_Toc500344550)

[5.8 SW-Component 12](#_Toc500344551)

[6. Documentation 12](#_Toc500344552)

[6.1 Annotation of Function Declarations 12](#_Toc500344553)

[6.2 Annotation of Variable Declarations 12](#_Toc500344554)

[6.3 Annotation of Control Structures 13](#_Toc500344555)

[6.4 Multiple-line Comments 13](#_Toc500344556)

[7. Testable Software 13](#_Toc500344557)

[7.1 Endless Loops 13](#_Toc500344558)

[7.2 Compiler-specific Settings 14](#_Toc500344559)

[7.2.1 Compiler Warnings 14](#_Toc500344560)

[7.2.2 Compiler-specific Problems 14](#_Toc500344561)

[7.2.3 Conditional Compilation 14](#_Toc500344562)

[7.2.4 Indenting of Conditional Compiling 14](#_Toc500344563)

[7.3 Other 15](#_Toc500344564)

[7.3.1 Masking Out of Bits 15](#_Toc500344565)

[7.3.2 Special Function Register (SFR) 15](#_Toc500344566)

[7.3.3 Filler Bytes / Alignment 16](#_Toc500344567)

[7.3.4 Interrupt Inhibits 17](#_Toc500344568)

[7.3.5 Macros for Access to Variables 17](#_Toc500344569)

[8. Examples 17](#_Toc500344570)

[9. Required Coding Rules 20](#_Toc500344571)

[9.1 MISRA-C: 2004 20](#_Toc500344572)

[10. Changes, References, Appendix, Terms 27](#_Toc500344573)

[10.1 References 27](#_Toc500344574)

# Coding guidelines constraints

To support the proper implementation, the coding guidelines for the modeling, or programming languages, shall address the topics listed in table below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TOPICS | | ASIL | | | |
| A | B | C | D |
| 1a | Use of language subsets b | ++ | ++ | ++ | ++ |
| 1b | Enforcement of strong typing c | ++ | ++ | ++ | ++ |
| 1c | Use of defensive implementation techniques | o | + | ++ | ++ |
| 1d | Use of style guides | + | ++ | ++ | ++ |
| 1e | Use of naming conventions | ++ | ++ | ++ | ++ |
| 1. The objectives of method 1b are:    * Exclusion of ambiguously defined language constructs which may be interpreted differently by different modelers, programmers code generators or compilers    * Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.    * Exclusion of language constructs which could result in unhandled run-time errors. 2. The objective of method 1c is to impose principles of strong typing where these are not inherent in the language. | | | | | |

# C Style Guide

This C Style Guide serves as a guideline for the software construction. The statements explained in this document shall be met.

## Upper and Lower Case

Many analysis tools run under UNIX or LINUX or use the build process in a CygWin environment. In contrast to Windows, file names and commands are case sensitive in that case. Therefore, take care to use correct spelling from the beginning.

## Encapsulation of SW-Modules and SW-Components

References of external resources (variables) must be avoided. The communication between SW-Modules and SW-Components is handled with functions and their parameters. Only with this practice can it be ensured, that SW-Modules and SW-Components are working independently from the environment and no side effects appear in the combination of SW-Modules and SW-Components.

## Minimal Function Amount

The function amount of a SW-Component shall be as minor as possible. So the SW-Component can be used in many different ways and the configuration amount will be minimized.

## Global Variables

Global variables should wherever possible only be used internally in a SW-Module. This will make integration of individual SW-Modules into other projects (encapsulation and reusability) easier. Initialization and definition should take place in the same C file.

## Missing Prototypes

When a project is compiled, the compiler option which outputs missing function prototypes as errors should be set (see documentation on the compiler used). A missing function prototype will always cause an error if the function parameter or return value is not of the type int. Without this compiler option, most compilers will not even give a warning. Since many sporadically occurring errors can be attributed to missing prototypes, prototypes should always be defined completely.

## Constants

### Realization of Constants with Enumerations

Constants should not be realized using the #define key word, as this causes the pre-processor to replace the constant identifier by the numerical value. When the emulator is used, it will display the numerical value which means that one may need to check what the value means.

The *enum* enumeration type should be used instead of realizing constants by means of the #define key word to ensure that the symbolic name will be displayed when an emulator is used.

### Comparison of Constants

In comparisons, the constant, always must be on the left side of the term. In this case the compiler would generate an error message, if the term contains a typing error which would change the meaning of the comparison into an assignment.

Example:

if (TRUE == INPUT\_AVAILABLE\_BL) /\* Comparison of a constant \*/

{ . . . }

## Defines

The #define pre-processor instruction should only be used for conditional compilation and for mathematical calculations with constants. Avoid redefinitions of function or variable identifiers. Where redefinition of an identifier seems to make sense, because this will make it easier to read, check if you have not chosen an inappropriate name from the start.

## Macros

In normal cases, macros should not be used, because their use can easily cause errors. The following example shows the problem by the definition of the macro abs(x) and the call of the macro:

Example:

#define abs(x) (((x)>0)?(x):-(x))

abs(a-b)

The pre-processor converts the call as follows.

Example:

(((a-b)>0)?(a-b):-(a-b))

The following example shows a macro using no parentheses, which execute the same command as the macro defined above.

Example:

#define abs(x) x>0?x:-x

abs(a-b)

Calling this macro the pre-processor converts the call as follows:

Example:

a-b>0?a-b:-a-b

This example shows that with missing parentheses the two macros are converted differently. Besides missing parentheses wrong parentheses and blanks can cause problems.

In established exceptional cases - e.g. reduction of runtime - macros can be used. Nesting of macros is not allowed because of missing clarity.

# Filenames

All C and header files used in the project must have different names. Using identical names is not allowed. This has to be considered in the software design. This convention shall be used consistently from design to the creation of test cases, because it helps to improve the traceability, readability and understand-ability.

The ending of a C file is .c. The use of .C is not allowed.

The ending of a header file is .h. The use of .H is not allowed.

# Header File

## Public / Private / Config Header

A header file describes the interface of exactly one SW-Module. Header files provide modularization and encapsulation.

At least one header file belongs to each C file. The public header file contains all constants, definitions of specific data types, function prototypes and other pre-processor directives, which are accessed externally. In reasonable exceptional cases variables can be declared in a header file. In general the access of variables by other SW-Modules and SW-Components is realized with access functions. This makes the declaration of variables not necessary in public header files.

The name of a public header file has to satisfy the following system: module name + “\_” + “public.h”, e.g. ilApplication\_public.h.

If there are declarations which are internally used for the C file, a private header file must be created. This private header file contains all input which only concerns the dedicated C file, e.g. variables. It can only be included by the dedicated C file.

The name of a private header file has to satisfy the following system: module name + “\_“ + “private.h”, e.g. ilApplication\_private.h.

If required, another header file for configuration can be used. The name of the configuration header file has to satisfy the following system: module name + “\_” + “cfg.h”, e.g. ilApplication\_cfg.h.

## Declarations and Definitions

It is not allowed to place statements which allocate storage such as definitions of variables into a header file. This would lead to multiple definitions of the same variable, because header files can be included in more than one C file. In header files variables can only be declared as external. The variable has to be defined in a central place in one C file (e.g. in the C file which contains the initialization of the variable). All other C files which are also using the variable have to include the header file, which declares the variable as extern.

Example:

main\_public.h

extern u8 main\_u8; /\* Variable is declared as extern, no memory will be allocated \*/

main.c

#include “main\_public.h” /\* Extern declaration of the variable is included \*/

u8 main\_u8; /\* Definition of the variable and allocating of memory \*/

main\_u8 = 0; /\* Initialization of the variable \*/

modul.c /\* This C file accesses the variable main\_u8 \*/

#include “main\_public.h” /\* Extern declaration of the variable is included \*/

if (main\_u8 ≥ 1) { . . .} /\* Access to the variable \*/

## Including Header Files

In a header file it is only allowed to use the pre-processor statement “#include” in reasonable exceptional cases, that means a header file does not include any other files.

Each header file must be provided with a protection against multiple inclusions (Include guard, see following example). The name of a header file is written in upper case letters and obtains the prefix “\_H”.

Example:

Header file: ilApplication\_public.h

#ifndef ILAPPLICATION\_PUBLIC\_H /\* For prevention of multiple includes \*/

#define ILAPPLICATION\_PUBLIC\_H

#endif

## Including Header Files into C Files

There are two possibilities for including header files into C files. Using the expression #include <modulname\_public.h>, the compiler looks for the header file first in the given include paths and afterwards relative to the C file. This kind of including is used for header files which belong to a library.

The second possibility is to include header files with the statement #include “modulname\_public.h”. In this case the compiler looks first relative to the C file and afterwards in the given include paths. This proceeding shows that the header file can be allocated to its own sources.

# Name Conventions

Below the name conventions for variables, constants, C functions, header files, C Modules and SW-Components are described. They must be used for external functions and symbols. The use for internal functions and symbols is recommended.

## Variables

These naming conventions are required for global variables. It is suggested to use them also for variables in function context. Each name of a variable contains the type of the variable, which determines the range. The following table lists all types which can be used:

|  |  |  |
| --- | --- | --- |
| Identifier | Abbreviation | Comment |
| u8 | u8 | 8-bit, unsigned |
| u16 | u16 | 16-bit, unsigned |
| u32 | u32 | 32-bit, unsigned |
| s8 | s8 | 8-bit, signed |
| s16 | s16 | 16-bit, signed |
| s32 | s32 | 32-bit, signed |
| float | flt | 32-bit, floating-point number |
| double | dbl | 64-bit, floating-point number |
| struct | st | Combination of different data types in a more complex one |
| enum | en | Data type whose set of values is a finite list of identifiers |
| union | un | Data structure, that stores several types of data in the same location |
| tag | tag | Name of a structure, enumeration and union |
| typedef | t | Can be used to give existing data types new names |

Each variable name is build up with the following informations:

|  |  |  |  |
| --- | --- | --- | --- |
| SW-Module name | Description | “\_” | Type |

The following table explains the structure of variables:

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Content | Example | Comment |
| SW-Module name | Short SW-Module name  3 – 5 letters | **diag**ErrCount\_u8 | The abbreviation of the SW-Module name shall be project-specific, uniform, and defined in the data dictionary. The SW-Module name may only contain characters which are part of the ISO C Standard (matches MISRA rules 5) and has to start with a lower case letter.  The use of the SW-Module name does not apply to local functions and variables. |
| Description | Meaningful description of the content | diag**ErrCount**\_u8 | The description is to be meaningful and may only contain identifiers, which are part of the ISO C Standard (matches MISRA rule 5). Each word of the description starts with an upper case letter. |
| \_ | Underscore | diagErrCount**\_**u8 | The underscore is not applicable in the case of a variable derived from a typedef |
| Type | Indicates the range of the variables | diagErrCount\_**u8** | The type is not applicable in the case of a variable derived from a typedef. It is only applicable for variables of type u8, u16, u32, s8, s16, s32, en, flt, dbl, st and tag.  If in the definition of a struct, enum or union a tag is declared, the tag name is the suffixed tag. |

The naming convention of pointer variables is similar to the one of variables. Between the underscore and the type of the pointer variable the name is extended with the letter ‘p’. The following definition creates a pointer to an u8 variable:

Example:

u8 \*diagErrValue\_pu8;

Furthermore the pointer operator \* has to be added always at the variable name. Using the following expression only the variable diagErrMinValue\_pu8 is declared as a pointer variable.

Example:

u8\* diagErrMinValue\_pu8, diagErrMaxValue\_pu8;

### Representation in Simplified Backus-Naur-Form (BNF)

<Variable name> ::= [<SW-Module name>]<Description>\_<Type>

<SW-Module name> ::= “Text without blanks“

<Description> ::= “Text without blanks“

<Type> ::= u8 | s8 | u16 | s16 | u32 | s32 | flt | dbl | st | un | en | tag

## Name Conventions for Fields in struct and union

The names of fields in struct or union follow the same convention as for the variables. Additionally, the following applies to bit fields: postfix bt (1 bit), u<x> (x > 1, for bit fields of type unsigned greater than 1 bit) and s<x> (x>1, for bit fields of type signed greater than 1 bit).

## Name Convention for typedef

All user-defined types, i.e. those defined with typedef, must end \_t.

## Name Convention for Constants

Constants here means the definitions for the pre-processor (via “#define“ or “enum”). In addition to the meaningful name requirement, the following two rules apply:

1. The names are written in capital letters.

2. Analogue to the name conventions for variables described in chapter 4, the type is also suffixed to the constant. The type is defined by the constant’s value range.

## Name Convention for Units (C Functions)

Function name must be different, not just in case and parameters, so misinterpretations and error is avoided.

The name of a C function must contain as a prefix an abbreviation of the SW-Component that it is part of and an identifier describing the functionality at the end. The following rule should be applicable to API functions as well as for internal functions.

If a C function is a handler, driver, manager or task function, then the following identification is inserted between the abbreviation of the SW-Component and the functionality identifier:

* Driver Drv
* Handler Hnd
* Manager Man
* Task Tsk
* Other <no identification>

Parameter and return value are regarded as variables, but don’t receive a SW-Component name prefix, see chapter 7.

Example:

error\_t adcDrvInit(adcProperty\_t \*property\_p) /\* Initialization of the driver ADC \*/

void adcDrvRead(channel\_t channel\_u8, u16 \*value\_pu16) /\* Reads a value from a channel \*/

## C Module

If a C module contains only one C function, the name of the C module must be identical with the name of the C function.2.4+1.6

In all other cases the name of a C module is a combination of the abbreviation of the dedicated SW-Component and an identifier describing the functionality, e.g. adcRead.c.

## Header File

The name of a header file consists of the abbreviation of the dedicated SW-Module name, “\_” and the key word “public” or “private”, e.g. adc\_public.h.

## SW-Component

If a SW-Component is a driver, handler, manager or task, the name contains the relative type token (see chapter 4.5) and the abbreviation of the description of the functionality. For example a driver which allows analog digital conversion is called DrvADC.

# Documentation

The comments must express the intention of the programmer. Each bigger code block must be explained with a comment which includes the reason for the creation of the code block.

The annotation has to be done in English.

It is only allowed to use C comments (e.g. /\* Comment \*/), the use of C++ comments (e.g. // Comment) is not allowed.

## Annotation of Function Declarations

Declarations of functions must be commented in detail in the dedicated header file, because they provide the interface of the SW-Module.

For creating test cases for the Unit Test a very well documented interface is important. The interface shall be documented with the following contents:

* Description of the functionality
* Input and output parameters (see chapter 5.2)

Example:

/\*========================================================\*/

/\* Description: \*/

/\* Input parameter: \*/

/\* Output parameter: \*/

/\*========================================================\*/

## Annotation of Variable Declarations

Declarations of variables have to be commented in detail in the dedicated header file, because they provide the interface of the SW-Module. Variables have to be documented with the following contents:

* Description (intended use of the variable)
* Value range [offset, error value, default value, initialization]
* Resolution (e.g. 20 mV/bit)
* Access possibility (e.g. read only, read/write)
* Updating rate (e.g. refresh every 20 ms)

Besides of variables also constants (#define) and data types are to be documented with a description.

Example:

/\* [Description][Value range][Resolution][Access possibility][Updating rate] \*/

## Annotation of Control Structures

Each control structure (If, switch, while, etc.) is to be commented. The comment must contain the intended use of the control structure.

Example:

/\* Description of this control structure \*/

This If statement is decision of

## Multiple-line Comments

Comments shall be started and finished in the same line. Some editors do not provide syntax highlighting, so that is not cognizable if a statement is ineffective because of having a comment introduced some lines above.

Example:

/\* Description A \*/

/\* Description B \*/

# Testable Software

The following chapter provides information on how to implement software in such a way that it can be tested with as little instruments as possible.

## Endless Loops

Some programs like Tessy and PolySpace terminate their analysis at endless loops. It is recommended to implement an endless loop in the following way.

Example:

do{

}

while (ENDLESS);

In this case ENDLESS is a macro that is defined in a header file as follows:

Example:

#ifdef TEST

extern unsigned char volatile ENDLESS

#else

#define ENDLESS 1

#endif

## Compiler-specific Settings

### Compiler Warnings

Compiler warnings are not to be ignored, as they are error messages based on a certain error detection threshold and therefore fully adequate error messages. They usually indicate a problem, which needs to be heeded and removed if possible.

The difference between an error and a warning is that referring to an error the object file won’t be generated, while a warning means that an object file is generated. But this object file will most probably not be in accordance with the specification.

Therefore, warnings need to be fixed wherever possible as being even more critical than errors.

### Compiler-specific Problems

As a rule, any processor-specific or compiler-specific instructions will make problems. Such instructions not only prevent automatic software testing, but also increase the amount of work for porting to other target systems. This includes pre-processor instructions such as #pragma and type qualifiers such as near, far and \_\_interrupt.

Since these instructions are compiler-dependent they should be avoided where possible. If no alternative is found, then the instructions should be implemented by macro definitions that can be masked out. Any compiler offers predefined constants for this purpose allowing you to identify whether or not that compiler is used.

A solution is the use of a central configuration header file in which these specific commands are expanded to empty instructions. The following example shows an excerpt of the configuration header file. The \_\_interrupt qualifier is not recognized by the GNU compiler and must therefore be masked out when this compiler is used.

Example:

/\* \_\_GNUC is predefined, developer needs not to define it \*/

#ifdef \_\_GNUC\_\_ /\* If the GNU compiler is used, \*/

#define \_\_interrupt /\* then \_\_interrupt is expanded to empty instruction \*/

#endif

### Conditional Compilation

Compiler options (#define) are often used for software variant generation. It is advisable to define these compiler options in a central configuration header file. Likewise, the effect of all compiler options is to be documented.

### Indenting of Conditional Compiling

Code that is compiled only conditionally should be indented to make it easier to identify as such.

Example:

#ifdef V850\_FJ2

#define MAX\_UART 2

#else

#define MAX\_UART 1

#endif

## Other

### Masking Out of Bits

Access operations by masks to individual bits should be avoided. The order of bits depends on implementation and compiler. Many compilers even allow selection. This makes the software difficult to port and automatic tests cannot be carried out. Additionally, the compiler compiles masking as a linear sequence of Read, Modify and Write instructions. As a consequence, the code is unnecessarily inflated and unpredictable results occur in the case of an interrupt. Therefore, bit arrays should preferably be defined as structures.

Example:

typedef struct

{

u16 lightIsOn\_bit : 1;

u16 engineRuns\_bit : 1;

u16 unused : 14;

}state\_t;

state\_t.lightIsOn\_bit = true;

That approach additionally has the advantage that the compiler performs the masking out and automatically always selects the right masking. Moreover, the name of the bit is displayed in the emulator (lightIsOn\_bit is more meaningful than bit2).

### Special Function Register (SFR)

The use of Special Function Registers is problematic, as they basically are variables which need to be linked to certain addresses. Not all compilers provide instructions for this.

Therefore, the SFR area should either be defined completely or as a divided structure. Moreover, a separate segment needs to be created to implement the structure.

The following example defines a controller with four port registers from address 0x1000 and two AD converter registers from address 0x2000:

Example:

sfrRegs\_public.h:

typedef struct PORTREGS{

u8 porta\_u8;

u8 portb\_u8;

u8 portc\_u8;

u8 portd\_u8;

}portregs\_t;

/\* insert safety check here if required (see below) \*/

typedef struct ADREGS{

u8 controlRegister\_u8;

u8 resultRegister\_u8;

}adregs\_t;

extern portregs\_t ports\_st ;

extern adregs\_t adRegister\_st ;

sfrRegs.c

#include<sfrRegs\_public.h>

#pragma section “PORTREGS” /\* depends on compiler, may differ \*/

portregs\_t ports\_st;

#pragma section “ADREGS” /\* depends on compiler, may differ \*/

adregs\_t adRegister\_st;

Additionally, two segments PORTREGS and ADREGS 0x1000 and 0x2000 need to be created in the linker file ( \*.ld ) from address 0x1000 and 0x2000. The segments should have absolute start values and sizes. A safety check can be installed in the include file to enable a size check.

Example:

#if 4\*sizeof(u8) != sizeof(PORTREGS)

#error “Please check size of struct PORTREGS.”

### Filler Bytes / Alignment

Some controller types expect variables which allocate 2 bytes in the memory (e.g. u16) to be located at addresses which can be divided by 2 and variables that allocate 4 bytes in the memory (e.g. u32) to be located at addresses which can be divided by 4.

Simple-structure variables of types u8, u16 or u32 and arrays of these types are automatically sorted by the linker. This creates a filler byte between the last u8 and the first u16 variables and two filler bytes between the last u16 variable and the fist u32 variable.

If variables are created in structures, then the linker cannot perform the sorting. In that case, alignment problems may result which cause unnecessary filler bytes. The example below shows a structure consisting of variables of type u8 and type u32.

Example:

typedef struct

{

u8 var1\_u8;

u32 var2\_u32;

}var\_t ;

All in all, the structure occupies 8 bytes in the memory although just 5 bytes are actually required. The reason for this is that 3 filler bytes are inserted before the u32 variable to ensure it is located at an address which can be divided by 4. The filler bytes can be saved if the structure is defined as shown in the example below:

Example:

typedef struct

{

u32 var2\_u32;

u8 var1\_u8;

}

In this case, no filler bytes are inserted between the u32 and the u8 variables, because an u8 variable may be located anywhere. Therefore, the structure occupies just 5 bytes in the memory.

Such manual sorting is a good way to save memory space and should therefore be used where possible when structures are declared.

Alignment problems may also occur where unions are used.

Example:

typedef union

{

u8 bytes\_u8[4];

struct byte\_st{

u8 firstbyte\_u8; u16 midbytes\_u16;

u8 lastbyte\_u8;

};

}var\_t;

In this case, the u8 array 4 occupies 4 subsequent bytes (byte 0, …, byte 3) in the memory. However, when the midbytes\_u16 element is accessed, the result will not be byte 1 and byte 2 as desired, but byte 2 and byte 3. The reason for this is that element midbytes\_u16 has to be located at an address which can be divided by 2. lastbyte\_u8 tries to access a 5th byte which does not exist according to array size.

The example below will likewise yield random results depending on array position.

Example:

u8 bytes\_u8[2];

u16 test\_u16 ;

test\_u16 = \*(u16\*)bytes\_u8;

Therefore, structures should be built sorted to optimize memory space. Unions should be structured in such a way that filler bytes are avoided.

Alignment and Struct-packing can also be adapted by #pragma instructions in justified exceptional cases.

### Interrupt Inhibits

Interrupt inhibits (ei/di) are frequently used to protect critical program parts from interruptions. Although many compilers have built-in functions, they do not save the preceding condition. For instance, the interrupt is only to be released after the processing of a critical section if it was released before the process. As the status register of a controller usually is not memory-mapped, own ASM functions or macros are used for the purpose. However, as ASM cannot be tested, the results are pseudo error messages.

The most reliable solution is the use of the functions provided by CASA which improve the portability of the source code in addition to testability.

### Macros for Access to Variables

Access operations to variables should always be effected directly and not by using macros, because macros are directly replaced by the pre-processor. Such replacement also takes place in the symbol table.

# Examples

The following examples explain the name conventions and the use of public and private header files.

**File: ilApplication\_private.h**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Include Check \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef ILAPPLICATION\_PRIVATE\_H /\* Prevents multiple integration of il\_private.h \*/

#define ILAPPLICATION\_PRIVATE\_H

#ifndef basetype\_h /\* Checks whether header file basetype.h is already included \*/

#error "please include basetype.h before including ilApplication\_private.h in your application"

#endif

#ifndef error\_h /\* Checks whether header file error.h is already included \*/

#error "please include error.h before including ilApplication\_private.h in your application"

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* General Definitions \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Interior lamp status \*/

#define IL\_LAMP\_ON\_U8 ((u8)240)

#define IL\_LAMP\_OFF\_U8 ((u8)15)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Local Variables \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern u8 interiorLampStatus\_u8;

extern boolean\_t interiorLampAlarm;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Internal Functions \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern error\_t getRxInteriorLightingInput(void);

extern void setInteriorLampRequest(u8 request\_u8);

#endif

**File: ilApplication\_public.h**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Include Check \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef ILAPPLICATION\_PUBLIC\_H /\* Prevents multiple integration of il\_public.h \*/

#define ILAPPLICATION\_PUBLIC\_H

#ifndef basetype\_h /\* Checks whether header file basetype.h is already included \*/

#error "please include basetype.h before including ilApplication\_public.h in your application"

#endif

#ifndef error\_h /\* Checks whether header file error.h is already included \*/

#error "please include error.h before including ilApplication\_public.h in your application"

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* General Definitions \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Watchdog channel \*/

#define IL\_TASK\_CHN\_U8 ((u8)0)

#define IL\_TASK\_ACT\_CHN\_U8 ((u8)1)

typedef enum DbfState\_t {

DBF\_LOW = 0, /\* Signal is on logical low level \*/

DBF\_HIGH /\* Signal is on logical high level \*/

} dbfState\_t;

typedef struct IL\_RX\_Data\_t{

dbfState\_t iL\_Fr\_Door\_Left\_State\_en;

dbfState\_t iL\_Fr\_Door\_Right\_State\_en;

dbfState\_t iL\_Bk\_Door\_Left\_State\_en;

dbfState\_t iL\_Bk\_Door\_Right\_State\_en; dbfState\_t iL\_Trunk\_Deck\_State\_en;

}ilRxData\_t;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Global Variables \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern ilRxData\_t ilRxData\_st;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* External Functions \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

extern void TASK\_InteriorLighting(void);

extern void InteriorLightingControl(void);

extern boolean\_t checkInteriorLightingInput(void);

extern void outputInteriorLightingData(void);

#endif

**File: ilApplication.c**

/\* Common includes \*/

#inlcude <basetype.h>

#include <error.h>

/\* Application specified includes \*/

#include <ilApplication\_private.h>

#include <ilApplication\_public.h>

void TASK\_InteriorLighting(void)

{ . . . }

void InteriorLightingControl(void)

{ . . . }

boolean\_t checkInteriorLightingInput(void)

{ . . . }

void outputInteriorLightingData(void)

{ . . . }

error\_t getRxInteriorLightingInput(void)

{ . . . }

void setInteriorLampRequest(u8 request\_u8)

{ . . . }

# Required Coding Rules

Actually MISRA-C standards are existing: MISRA-C:2004. In general the MISRA standard MISRA-C:2004 has to be used.

In the tables below, automatically checkable rules are marked with a “√“. Rules which need to be checked

in a manual review are marked with “☞“.

## MISRA-C: 2004

The following three defined categories contain a subset of MISRA-C: 2004 Rules. The category

“Exceptional” contains the smallest subset, “HIS” contains all MISRA-C:2004 Rules which are required by the “Hersteller Iniviative Sofware” (HIS) and “Auto generated TL” contains all MISRA-C:2004 Rules which can be met in general by auto generated code from TargetLink.

| **MISRA-C:2004 Rules** | **Category** | | |
| --- | --- | --- | --- |
| **Exceptional** | **HIS** | **Auto generated TL** |
| **[Rule 1.1]** All code shall conform to ISO 9899:1990 C programming language, ISO  9899, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC  9899/AMD1:1995, and ISO/IEC 9899/COR2: 1996. |  | √ | √ |
| **[Rule 1.2]** No reliance shall be placed on undefined or unspecified behaviour. |  | √ | √ |
| **[Rule 1.3]** Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform. |  | ☞ |  |
| **[Rule 1.4]** The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers. |  | ☞ |  |
| **[Rule 1.5]** Floating-point implementations should comply with a defined floating point standard. |  | ☞ |  |
| **[Rule 2.1]** Assembly language shall be encapsulated and isolated. |  | √ | √ |
| **[Rule 2.2]** Source code shall only use C-style comments. |  | √ | √ |
| **[Rule 2.3]** The character sequence /\* shall not be used within a comment. |  | √ | √ |
| **[Rule 2.4]** Sections of code should not be 'commented out'. |  | ☞ |  |
| **[Rule 3.1]** All usage of implementation-defined behaviour shall be documented. |  | √ |  |
| **[Rule 3.2]** The character set and the corresponding encoding shall be documented. |  | ☞ |  |
| **[Rule 3.3]** The implementation of integer division in the chosen compiler should be determined, documented and taken into account. | ☞ | ☞ | ☞ |
| **[Rule 3.4]** All uses of the #pragma directive shall be documented and explained. |  | √ |  |
| **[Rule 3.5]** If it is being relied upon, the implementation-defined behaviour and packing of bitfields shall be documented. |  | ☞ |  |
| **[Rule 3.6]** All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation. |  | √ | √ |
| **[Rule 4.1]** Only those escape sequences that are defined in the ISO C standard shall be used. |  | √ | √ |
| **[Rule 4.2]** Trigraphs shall not be used. |  | √ | √ |
| **[Rule 5.1]** Identifiers (internal and external) shall not rely on the significance of more than 31 characters. |  | √ | √ |
| **[Rule 5.2]** Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. |  | √ | √ |
| **[Rule 5.3]** A typedef name shall be a unique identifier. |  | √ | √ |
| **[Rule 5.4]** A tag name shall be a unique identifier. |  | √ | √ |
| **[Rule 5.5]** No object or function identifier with static storage duration should be reused. |  | √ | √ |
| **[Rule 5.6]** No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names. |  | √ | √ |
| **[Rule 5.7]** No identifier name should be reused. |  | ☞ |  |
| **[Rule 6.1]** The plain char type shall be used only for the storage and use of character values. |  | √ | √ |
| **[Rule 6.2]** Signed and unsigned char type shall be used only for the storage and use of numeric values. |  | √ | √ |
| **[Rule 6.3]** Typedefs that indicate size and signedness should be used in place of the basic types. |  | √ | √ |
| **[Rule 6.4]** Bit fields shall only be defined to be of type unsigned int or signed int. |  | √ | √ |
| **[Rule 6.5]** Bit fields of type signed int shall be at least 2 bits long. |  | √ | √ |
| **[Rule 7.1]** Octal constants (other than zero) and octal escape sequences shall not be used. |  | √ | √ |
| **[Rule 8.1]** Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call. | √ | √ | √ |
| **[Rule 8.2]** Whenever an object or function is declared or defined, its type shall be explicitly stated. |  | √ | √ |
| **[Rule 8.3]** For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical. |  | √ | √ |
| **[Rule 8.4]** If objects or functions are declared more than once their types shall be compatible. |  | √ | √ |
| **[Rule 8.5]** There shall be no definitions of objects or functions in a header file. |  | √ | √ |
| **[Rule 8.6]** Functions shall be declared at file scope. |  | √ | √ |
| **[Rule 8.7]** Objects shall be defined at block scope if they are only accessed from within a single function. |  | √ | √ |
| **[Rule 8.8]** An external object or function shall be declared in one and only one file. |  | √ | √ |
| **[Rule 8.9]** An identifier with external linkage shall have exactly one external definition. |  | √ | √ |
| **[Rule 8.10]** All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. |  | √ | √ |
| **[Rule 8.11]** The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage. |  | √ | √ |
| **[Rule 8.12]** When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation. |  | √ | √ |
| **[Rule 9.1]** All automatic variables shall have been assigned a value before being used. | √ | √ | √ |
| **[Rule 9.2]** Braces shall be used to indicate and match the structure in the nonzero initialisation of arrays and structures. | √ | √ | √ |
| **[Rule 9.3]** In an enumerator list, the '=' construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised. |  | √ | √ |
| **[Rule 10.1]** The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same ignedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression. |  | √ |  |
| **[Rule 10.2]** The value of an expression of floating type shall not be implicitly converted to a different type if: a) it is not a conversion to a wider floating type, or b) the expression is complex, or c) the expression is a function argument, or d) the expression is a return expression. |  | √ | √ |
| **[Rule 10.3]** The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression. |  | √ |  |
| **[Rule 10.4]** The value of a complex expression of floating type may only be cast to a narrower floating type. |  | √ | √ |
| **[Rule 10.5]** If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand. | ☞ | ☞ |  |
| **[Rule 10.6]** A "U" suffix shall be applied to all constants of unsigned type. |  | √ |  |
| **[Rule 11.1]** Conversions shall not be performed between a pointer to a function and any type other than an integral type. |  | √ | √ |
| **[Rule 11.2]** Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void. |  | √ | √ |
| **[Rule 11.3]** A cast should not be performed between a pointer type and an integral type. |  | √ | √ |
| **[Rule 11.4]** A cast should not be performed between a pointer to object type and a different pointer to object type. |  | √ |  |
| **[Rule 11.5]** A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer. |  | √ |  |
| **[Rule 12.1]** Limited dependence should be placed on C's operator precedence rules in expressions. |  | √ |  |
| **[Rule 12.2]** The value of an expression shall be the same under any order of evaluation that the standard permits. | √ | √ | √ |
| **[Rule 12.3]** The sizeof operator shall not be used on expressions that contain side effects. |  | √ | √ |
| **[Rule 12.4]** The right hand operand of a logical && or || operator shall not contain side effects. |  | √ | √ |
| **[Rule 12.5]** The operands of a logical && or || shall be primary-expressions. |  | √ | √ |
| **[Rule 12.6]** The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, || and !). |  | √ |  |
| **[Rule 12.7]** Bitwise operators shall not be applied to operands whose underlying type is signed. | √ | √ |  |
| **[Rule 12.8]** The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand. | √ | √ |  |
| **[Rule 12.9]** The unary minus operator shall not be applied to an expression whose underlying type is unsigned. |  | √ |  |
| **[Rule 12.10]** The comma operator shall not be used. |  | √ | √ |
| **[Rule 12.11]** Evaluation of constant unsigned integer expressions should not lead to wrap-around. | √ | √ | √ |
| **[Rule 12.12]** The underlying bit representations of floating-point values shall not be used. |  | √ | √ |
| **[Rule 12.13]** The increment (++) and decrement (--) operators should not be mixed with other operators in an expression. |  | √ |  |
| **[Rule 13.1]** Assignment operators shall not be used in expressions that yield a Boolean value. | √ | √ | √ |
| **[Rule 13.2]** Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. |  | √ | √ |
| **[Rule 13.3]** Floating-point expressions shall not be tested for equality or inequality. | √ | √ |  |
| **[Rule 13.4]** The controlling expression of a for statement shall not contain any objects of floating type. | √ | √ | √ |
| **[Rule 13.5]** The three expressions of a for statement shall be concerned only with loop control. |  | √ | √ |
| **[Rule 13.6]** Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop. |  | √ | √ |
| **[Rule 13.7]** Boolean operations whose results are invariant shall not be permitted. |  | √ | √ |
| **[Rule 14.1]** There shall be no unreachable code. | √ | √ | √ |
| **[Rule 14.2]** All non-null statements shall either (i) have at least one side-effect however executed, or (ii) cause control flow to change. | √ | √ | √ |
| **[Rule 14.3]** Before pre-processing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character. |  | √ |  |
| **[Rule 14.4]** The goto statement shall not be used. |  | √ | √ |
| **[Rule 14.5]** The continue statement shall not be used. |  | √ | √ |
| **[Rule 14.6]** For any iteration statement there shall be at most one break statement used for loop termination. |  | √ | √ |
| **[Rule 14.7]** A function shall have a single point of exit at the end of the function. |  | √ | √ |
| **[Rule 14.8]** The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement. |  | √ | √ |
| **[Rule 14.9]** An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement. |  | √ | √ |
| **[Rule 14.10]** All if ... else if constructs shall be terminated with an else clause. | √ | √ |  |
| **[Rule 15.1]** A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement. | √ | √ | √ |
| **[Rule 15.2]** An unconditional break statement shall terminate every non-empty switch clause. | √ | √ | √ |
| **[Rule 15.3]** The final clause of a switch statement shall be the default clause. | √ | √ |  |
| **[Rule 15.4]** A switch expression shall not represent a value that is effectively  Boolean. |  | √ | √ |
| **[Rule 15.5]** Every switch statement shall have at least one case clause. |  | √ | √ |
| **[Rule 16.1]** Functions shall not be defined with a variable number of arguments. |  | √ | √ |
| **[Rule 16.2]** Functions shall not call themselves, either directly or indirectly. |  | √ | √ |
| **[Rule 16.3]** Identifiers shall be given for all of the parameters in a function prototype declaration. |  | √ | √ |
| **[Rule 16.4]** The identifiers used in the declaration and definition of a function shall be identical. | √ | √ | √ |
| **[Rule 16.5]** Functions with no parameters shall be declared with parameter type void. |  | √ | √ |
| **[Rule 16.6]** The number of arguments passed to a function shall match the number of parameters. | √ | √ | √ |
| **[Rule 16.7]** 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. |  | √ | √ |
| **[Rule 16.8]** All exit paths from a function with non-void return type shall have an explicit return statement with an expression. | √ | √ | √ |
| **[Rule 16.9]** A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty. |  | √ | √ |
| **[Rule 16.10**] If a function returns error information, then that error information shall be tested. |  | √ | √ |
| **[Rule 17.1]** Pointer arithmetic shall only be applied to pointers that address an array or array element. |  | ☞ |  |
| **[Rule 17.2]** Pointer subtraction shall only be applied to pointers that address elements of the same array. |  | ☞ |  |
| **[Rule 17.3]** >, >=, <, <= shall not be applied to pointer types except where they point to the same array. |  | ☞ |  |
| **[Rule 17.4]** Array indexing shall be the only allowed form of pointer arithmetic. |  | √ |  |
| **[Rule 17.5]** The declaration of objects should contain no more than 2 levels of pointer indirection. |  | √ | √ |
| **[Rule 17.6]** The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. |  | √ | √ |
| **[Rule 18.1]** All structure and union types shall be complete at the end of a translation unit. |  | √ | √ |
| **[Rule 18.2]** An object shall not be assigned to an overlapping object. |  | ☞ |  |
| **[Rule 18.3]** An area of memory shall not be reused for unrelated purposes. |  | ☞ |  |
| **[Rule 18.4]** Unions shall not be used. |  | √ |  |
| **[Rule 19.1]** #include statements in a file should only be preceded by other preprocessor directives or comments. |  | √ | √ |
| **[Rule 19.2]** Non-standard characters should not occur in header file names in #include directives. |  | √ | √ |
| **[Rule 19.3]** The #include directive shall be followed by either a <filename> or "filename" sequence. |  | √ | √ |
| **[Rule 19.4]** C macros shall only expand to a braced initialiser, a constant, a parenthesised expression, a type qualifier, a storage class specifier, or a do-whilezero construct. |  | √ | √ |
| **[Rule 19.5]** Macros shall not be #define'd or #undef'd within a block. |  | √ | √ |
| **[Rule 19.6]** #undef shall not be used. |  | √ | √ |
| **[Rule 19.7]** A function should be used in preference to a function-like macro. |  | √ |  |
| **[Rule 19.8]** A function-like macro shall not be invoked without all of its arguments. | √ | √ | √ |
| **[Rule 19.9]** Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. |  | √ | √ |
| **[Rule 19.10]** In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##. | √ | √ |  |
| **[Rule 19.11]** All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator. | √ | √ | √ |
| **[Rule 19.12]** There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition. |  | √ | √ |
| **[Rule 19.13]** The # and ## preprocessor operators should not be used. |  | √ | √ |
| **[Rule 19.14]** The defined preprocessor operator shall only be used in one of the two standard forms. |  | √ | √ |
| **[Rule 19.15]** Precautions shall be taken in order to prevent the contents of a header file being included twice. |  | √ | √ |
| **[Rule 19.16]** Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor. |  | √ | √ |
| **[Rule 19.17]** All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related. |  | √ | √ |
| **[Rule 20.1]** Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined. | √ | √ | √ |
| **[Rule 20.2]** The names of standard library macros, objects and functions shall not be reused. | √ | √ | √ |
| **[Rule 20.3]** The validity of values passed to library functions shall be checked. |  | ☞ |  |
| **[Rule 20.4]** Dynamic heap memory allocation shall not be used. |  | √ | √ |
| **[Rule 20.5]** The error indicator errno shall not be used. |  | √ | √ |
| **[Rule 20.6]** The macro offsetof, in library <stddef.h>, shall not be used. |  | √ | √ |
| **[Rule 20.7]** The setjmp macro and the longjmp function shall not be used. |  | √ | √ |
| **[Rule 20.8]** The signal handling facilities of <signal.h> shall not be used. |  | √ | √ |
| **[Rule 20.9]** The input/output library <stdio.h> shall not be used in production code. |  | √ | √ |
| **[Rule 20.10]** The library functions atof, atoi and atol from library <stdlib.h> shall not be used. |  | √ | √ |
| **[Rule 20.11]** The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used. |  | √ | √ |
| **[Rule 20.12]** The time handling functions of library <time.h> shall not be used. |  | √ | √ |
| **[Rule 21.1]** Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults. |  | √ | √ |

# Changes, References, Appendix, Terms

## References

|  |  |  |
| --- | --- | --- |
| **Category** | **Document Name** | **Document Number** |
| Process | C Style Guide | AD-PE1-1-23 |
| Process | SW Construction | HP-GE-232-12 |