Coding guidelines

1. (C, C++, mandatory) Language
   The source code shall be written in English (comments, variable- and function-names, etc.).
2. (C, C++, mandatory) goto
   goto shall not be used in the code.
3. (C, C++, mandatory) Number of executable lines per function
   Each function shall not have more than 200 executable lines of code.
4. (C, C++, advisory) Code lines per file
   Each source file should be smaller than 750 NLOC (Non commentary lines of code).
5. (C, -, required) global variables
   The usage of global variables should be avoided.
6. (-, C++, required) extern keyword
   Extern declarations should be avoided.
7. (-, C++, required) Access Specifier
   Use the least visibility as possible to get slim interfaces and to fulfill the principle of information hiding.
8. (C,-,mandatory) Location of variable declaration
   Variables shall only be declared at the start of a compound statement, not anywhere in a compound. If a const "variable" is needed somewhere within a compound statement, you can open another scope.
9. (C, C++, mandatory) Usage of low level types
   The following derived data types shall be used. No C standard data types (int, char, etc.) shall be used. sbit8, ubit8, sbit16, ubit16, sbit32, ubit32: One bit in a bit field of size [1..8], [9..16], [17..32]:
   a. sint8, uint8: 8 bit values
   b. sint16, uint16: 16 bit values
   c. sint32, uint32: 32 bit values
   d. sint64, uint64: 64 bit values
   e. float32, float64: Floating point values
   f. boolean values:
   g. For C: boolean with b_FALSE and b_TRUE
   h. For C++: bool with false and true
10. (C, advisory, C++, mandatory) const vs. #define
    Constant values should be declared via a const declaration and not via the #define statement where possible.
11. (C, C++, mandatory) const in C vs. C++
    Defining constants at global- or namespace scope in header files is allowed but only as static const (i.e. no const without static or extern keyword).
12. (C, C++, required) magic numbers
    Symbolic names for constant values are mandatory. Do not use hard coded (magic) numbers directly in expressions. The symbolic name should be defined only once in an appropriate scope (for example in a header file if it belongs to an interface).
    Defines that only have the corresponding number as a word in the name (e.g. DEF_ONE or DEF_ZERO) are not allowed!
13. (C, C++, required) Returning structs from functions
    Object and structs shall not be returned by value by functions or operators.
14. (C, C++, required) Large objects on stack
    Large objects, structs or arrays shall not be placed on the stack.

15. (C, C++, required) Function calls inside conditions

    Functions shall not be called from inside branch- and loop conditions or in function call arguments.

```
// discouraged
Bread myBread = bake(knead(mix(flour, water, salt, soda)));

// encouraged
IngredientMixture myLittlePile = mix(flour, water, salt, soda);
Dough             myDough      = knead(myLittlePile);
Bread             myBread      = bake(myDough);
```

16. (C, C++, mandatory) Indentation

    Two spaces shall be used for indentation. No tab characters are allowed. The editor shall be configured to replace tabs with spaces.

17. (C, C++, mandatory) Code Style

    Allman style shall be used for block braces (http://en.wikipedia.org/wiki/Indent_style#Allman_style)

18. (C, C++, required) Preprocessor Alignment

    Nested preprocessor directives shall be indented by adding two spaces per indent level in front of the "#" character accordingly. Preprocessor indentation does not influence code indentation, nor the other way round.

```
#ifdef BLA
  #ifdef FOO
    doThis();
  #else
    doThat();
  #endif
#else
    // do nothing
#endif
```

19. (C, C++, required) Vertical Alignment

    Vertically align blocks in semantically similar lines for better readability. Also align punctuation marks in those lines.

```
Something foo = {
                  { umpa_lumpa::chocolate, &umpa_lumpa::produce       },
                  { lala_land::rainbow   , &lala_land::unicorn::shoot },
                  { dream_land::dreams   , &dream_land::sleep         }
                };
uint32         bar            = 5;
const uint8    spam           = 8;
MyUberEnumType superDuperEnum = bestEnumValueEver;
```

20. (C, C++, required) Spacing for type modifiers

    Type modifiers like & (reference) and * (pointer) shall be attached to the type rather than to the declared

identifier.

```
void* bla;  // YES
void *bla;  // NO
void * bla; // NO

// This is especially valid for such cases with different type name lengths:

uint32*      bla; // YES
LongTypeName foo;

uint32      *bla; // NO
LongTypeName foo;
```

21. (C, C++, required) Blank Lines

    Not more than one or maximum two consecutive blank lines should be used in the code.

22. (C, C++, mandatory) Spacing around Operators

    At least one space shall be used on each side of binary operators (binary as in "takes two arguments"), except the ->-operator and the .-operator. There shall be no space between unary operators and operands.

```
a = &b + *p_c;       // YES
a=& b+* p_c;         // NO NO NO
if(p_a->b > p_c->d) // YES
if(p_a->b>p_c->d)   // NO
```

23. (C, C++, mandatory) Trailing Whitespace

    There shall be no trailing whitespace.

24. (C, C++, mandatory) Newline at File End

    Files shall end with a newline.

25. (C, advisory, C++, mandatory) /* comments

    Multi-line comments (/* ... */) shall not be used. Use single line comments (//) instead.

26. (C, C++, mandatory) Line Continuation in Comments

    Line continuations for one-line comments are not allowed.

```
// blabla, my comment \
Against contrary beliefs, this is actually still comment, which is confusing!
\
Even the confluence code highlighting is confused.
```

27. (C, C++, mandatory) Naming Overview

Denomination	Explanation	Applicability	Examples
all lowercase	Numbers and lower case letters and underscores as word separators → no leading/trailing underscore, not double underscores	• namespaces • file names	my_fantastic_name unicorn
all uppercase	Numbers and upper case letters and underscores as word separators → no leading/trailing underscore, not double underscores	• defines • not constants	MY_FANTASTIC_NAME UNICORN

upper CamelCase	Numbers and letters, CamelCase for word separation, starting with a capital letter	• types (except POD types)
		  ○ classes
		  ○ enum Types
		  ○ structs
		  ○ unions
		  ○ ...
			MyFantasticName
			Unicorn

lower camelCase	Numbers and letters, camelCase for word separation, starting with a lowercase letter	• variables (regardless if stack or static) • constants • enum constants and variables • struct / Class Members • function Parameters • methods / Functions (except con/destructors)	myFantasticName unicorn

28. (C, C++, mandatory) File Names

    Filenames use the "all lowercase" pattern. Optionally there can be a module or sub-module prefix, but do not repeat the folder structure as a filename-prefix. The filename should have a descriptive name of its content.

29. (C, C++, mandatory) Namespace and Module Prefix/Unit Prefix

    All global identifiers (this also applies to enum values) shall be in a namespace. In cases where namespaces are not available (C, #defines (also in C++)), a module prefix shall be used instead. Class scopes in C++ serve as namespaces for this purpose as well.

```
void BAR_bestFunctionInTheWorld();
// or even
void BARFOO_bestFunctionInTheWorld();
// if FOO is a part of BAR and the differentiation is intended and the
combination of "BARFOO" doesn't become too long.


// There are no namespaces in C, so we have to use prefixes.
typedef enum
{
  BARFOO_apple,
  BARFOO_banana,
  BARFOO_meat,
  BARFOO_salad
} BARFOO_Food;
```

30. (C, C++, mandatory) Mandatory Prefixes

    p_ has to be used as a prefix for pointer types. The number of ps indicates the number of pointer indirections.

31. (C, C++, mandatory) Extended Prefix Definition

Prefix	Type
a	array
r	reference
p	pointer
b	bool / boolean
u	unsigned
s	signed
f	any float (also double)
v	void
e	variable of type "enum" (not the enum constants, nor the enum type!)
t	types in general, e.g. typedefs, structs, unions, but not classes, also not the instances of these types.

When extended prefixes are used, functions get a prefix that indicates their return type additionally to what is

required by the mandatory prefixes (separated by an underscore).

```
void v_fooBar(void** pp_stuff);


// "p" because it *is* a pointer and "v" because it returns void.
void (*)(void** pp_arg) p_v_myFunction;
p_v_myFunction = &v_fooBar;


uint8* (*)(void) p_p_myOtherFunction; // first p because it *is* a pointer,
second p because it returns a pointer.
```

32. (C, C++, required) self-contained header files

    Header files should be self-contained, i.e. a c(pp)-file that only includes one single header file shall compile out of the box without having to include other header files before. Besides, a module shall not rely on indirectly included header files.

33. (C, C++, required) minimal includes

    #includes should be minimal as in "Only include what you need", i.e. avoid unnecessary includes.

34. (C, C++, mandatory) Braces in Macros

    The content as well as the arguments of a macro shall always be enclosed in braces (if possible). Use a meaningful name for each parameter of a function like macro.

```
// NOT OK:
#define ADD_BAD(x, y) \
  x + y


// OK:
#define ADD_GOOD(x, y) \
  ((x) + (y))


// NOT OK (because of the arguments):
#define SYS_GET_ALARM_BASE(x, y) \
  (GetAlarmBase(x, y))


// OK:
#define SYS_GET_ALARM_BASE(alarmId, info) \
  ((uint32)(((alarmId) * 2) + WARNING_OFFSET + (info)))
```

35. (C, C++, advisory) Safe Precompiler Switches

    To ensure that the corresponding preprocessor define has actually been defined the following macro should be used. Some compilers do not even provide a warning when a preprocessor define is used (evaluated) without

being set.

```
// The following defines ensure, that the config-switch-defines exist.
// You will get a syntax error if you forgot to include the module's config file
// or if you didn't define the values properly.
#define CFG_ON        (1
#define CFG_OFF       (0
#define CFG_CHECK(X)   X)

// In the header
// Set the preprocessor define
#define MDL_MY_SUPER_FEATURE CFG_ON

// In the source file
// Use the preprocessor define
#if CFG_CHECK(MDL_MY_SUPER_FEATURE)
// Code for super feature goes here.
// Like this, it is ensured that MDL_MY_SUPER_FEATURE has been defined as either CFG_ON or CFG_OFF.
#endif
```

36. (C, C++, mandatory) Naming Configuration Files

    A module's configuration file shall be named <module_name>_cfg.h. The same rule applies to components and units respectively.

37. (C, C++, required) Config files should not be implementation files

    A configuration file for a module should be a header file only if possible. Implementation files (C-/C++-files) should be avoided.

    Example

    my_module_cfg.c ← discouraged

    my_module_cfg.cpp ← discouraged

    my_module_cfg.h ← recommended

38. (C, C++, required) Usage of correct single precision suffix for float constants

    All floating point constants shall be defined numerically (i.e. not as a calculation) with the correct qualifier, "F" or "f" for single precision. It is recommended to use the capital "F".

    Example

    ```
    #define PI_OVER_SIX    (CML_PI / 6.F)   // Wrong:    Conditions of evaluation unclear
    #define PI_OVER_SIX    (0.52359878F)    // Correct: Always has the same value regardless of the compiler
    ```

39. (C, C++, required) Prefer multiplication over division for constants

    Floating point divisions by constants shall not be performed. Instead, a multiplication by its reciprocal shall be performed.

    ```
    float32 c4 = c3 * 0.01666666666F; // Exact single precision float guaranteed.
    float32 c4 = c3 / 60.0F;          // Usually optimized to multiplication, but precision not guaranteed to be the same.
    ```

40. (C, C++, required) Avoid multiple float operations in one statement

    The result of every individual floating point operation, including type casts to a floating point type must be

explicitly assigned to a variable.

Example

```
// Wrong
float32 result = 0.5F * y * (CML_PI - (y2 * (c1 - y2 * c2)));

// Right
float32 temp;
float32 result;
temp = y2 * c2;
temp = c1 - temp;
temp = y2 * temp;
temp = CML_Pi - temp;
temp = y * temp;
result  = 0.5F * temp;
```

41. (C, C++, required) Code Comments

    There shall be a space between // and the text.