HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

Checklist

Compliance with coding guidelines

Basis:

Coding guidelines „The Power of 10“
(Holzmann, Gerard J.: „The Power of 10: Rules for Developing Safety - Critical Code”; NASA/JPL Laboratory for Reliable Software; 2006)

Project Name (Software):	Safe Temperature Monitoring
Filename:	monitorTemp.c
Name of the Software Module/Software Function (s)	MonitorTemp
Creation Date :	25-06-2023
Author:	CKN
Status (Draft, Verified, Approved):	Draft
Final approval of the document	
25-06-2023 Date	Kaushiknarayanan Chandrasekaran Name (Plain text) / Signature

Versions:

Edited:	14.04.20/sch	27.04.20/sch	14.05.20/sch	01.04.21/sch	25.06.23/ckn				
Checked:									
Rev/Version:	0.1	0.2	0.3	0.4	0.5				

(For details see chapter 4)

1. Content

2. Table (checklist) coding guidelines

Nr.	Rule	Implemented (Yes/No)	Remarks (including date)
1	Restriction to simple control flows (no goto, setjmp, longjmp instructions, no recursion), application of the structured programming paradigm; Also note additional coding style: Provide modules/functions with a comment head, format the source code uniformly (indentation, use of meaningful identifiers, open curly brackets in the same line, no block comments except in the comment head, always only one instruction per line, detailed comments in the source code etc., see also Section 3 and current common guidelines such as "(C-)coding conventions, tips and tools, Version 3.2, 07/06/2009, T. Birnthaler, OSTC GmbH"	Yes	25-06-2023: Forbidden statements like 'goto' functions are not implemented in the 'Safe Temperature Monitoring' project
2	Loops must have an upper and clearly defined limit for the number of cycles, no use of endless loops (exception: while (1) loop as the "main loop" in microcontroller programming)	Yes	25-06-2023: No infinite loops in the safety function have been implemented
3	After initialization no dynamic memory allocation shall be carried out (no use of the malloc() instruction) in order to avoid that more memory at runtime is reserved than physically available.	Yes	25-06-2023: No memory related operations handled in the safety function
4	Each function must be limited in its size, if possible the number of code lines (without comments) should not exceed 60 (corresponds to approx. one A4 page); C source code files that contain several functions should not be excessively large (max. approx. 1000 lines), instead source code file shall be split into several files.	Yes	25-06-2023: Safety function has been implemented as per 'Single Responsibility Principle'.
5	Use of "Assertion Programming" technique (checking value ranges/applying plausibility checks) and defensive programming techniques, if possible at least two checks per function; in particular	Yes	25-06-2023: Plausibility checks are properly implemented in the safety function

	checking of function parameters/intermediate results/return values for plausibility (e.g. do not divide by 0) and for valid value ranges as well as verifying and limiting data access		
6	Variables/data must always be declared on the lowest level of scope, i.e. use the principle of limited access (encapsuling) if possible (define variables locally and not globally); Define global variables only in exceptional cases (if possible at one central point in source code, so that all programmers can overview this), use of global variables must be justified.	Yes	25-06-2023: Local variables are properly handled inside the safety function
7	When calling a function with a return value, this value must be checked for correctness/ plausibility. If parameters are passed to a function, these parameters must be checked for correctness/plausibility within the function (Note: Rule No. 7 complements/tightens Rule No. 5 particularly with regard to functions, but Rule No. 5 may require additional checks of intermediate values or memory access).	Yes	25-06-2023: The return values are evaluated before doing further operations in the safety function.
8	The use of preprocessor instructions must be limited to a minimum (integration of header files, definition of simple macros, constants etc.); complex constructs e.g. for conditional compilation are not allowed, as this creates a large number of (and therefore fault-prone) compilation variants.	Yes	25-06-2023: Preprocessor macros are only used for constant values which is desirable.
9	Pointers may only be used in exceptional cases, these exceptions must be justified; multiple referencing should be avoided; Dereferencing should never be hidden in macro or type definitions.	Yes	25-06-2023: No pointers were used in the safety function
10	The source code must be compiled with the strictest warning setting; all warnings must be analyzed and resolved; after the final compilation, there should be no more warnings.	Yes	25-06-2023: Source code doesn’t have warnings.

3. Overview of formatting and coding rules

The following formatting and coding rules should be considered when creating source text. The aim is to ensure the readability, verifiability, reliability, maintainability, expandability and reusability of the software code.

Source: Based on *„(C-)Programmier-Konventionen, Tipps und Werkzeuge*, Version 3.2, 06.07.2009, T. Birnthaler, OSTC GmbH“ (in excerpts and each adapted)

Structure of the C source code file

The existing templates (HRW/Functional Safety Department/Prof. Schepers) shall be applied for C source code files and the associated header files!

Structure of the source code files:

- File comment header with the following information (if applicable):
  - Module name
  - File name (possibly also associated header file)
  - Creation date
  - Author
  - Developer group
  - Project
  - Copyright notice (if necessary)
  - Dependencies on other libraries / source code files
  - Brief description of the content and functionality
  - List of changes with the following information (for each change):
    Version / date / author / reason for the change (latest change at the top)
- #include instructions <...> for system header files
- # include instructions "..." for your own header files
- # define statements (constants and macros) must be done in the header!
- Definition of data types (typedef)
- Definition of global constants (const)
- Definition of global variables
- Definition of module local constants (static const)
- Definition of module local variables (static)
- Declaration of module local functions (static)
- Definition of global and module local functions (meaningfully sorted by topic)

Structure of the header files:

- Include header declarations with preprocessor directives "#ifndef, #define and #endif" (see template for headers) in order to avoid errors with multiple integration
- #define statements (constants and macros)
- Declaration of data types (typedef)
- Declaration of global constants (extern const)
- Declaration of global variables (extern)
- Declaration of global functions (extern)

Hints:

Functions must be fully declared so that a compiler check is possible. Declarations of global variables and functions (i.e. declaration as “extern”) must be exported to a separate header file and this must be included in all source files with "include". Module local variables and functions used exclusively in the source file itself must be declared as "static" and must not be included in an external header file.

It makes sense to create a bug list and, if necessary, a to-do list for a software module or project.

Assignment of names for functions, data types, variables and constants

- Names should be chosen in such a way that they ensure an understandable source code.
- The naming must be carried out according to the following system:

	1. Letter	Subsequent letters	Underline	Examples
Constants Macros Data types	Capital letter Capital letter Capital letter	Capital letter Capital letter Capital + Lowercase letter	Yes Yes No	PI, ARRAY_LEN, COLOR MULTIPLY, COLLECT_DATA Address, CustomerAdress
Variables	Lowercase	Lowercase	Yes	number, iban_number
Functions	Capital letter	Capital + Lowercase letter	No	PrintFile, CalculateCRC

- Functions must contain a verb such as: Get, Init, Delete etc. In addition to the action word, functions can (should) combine a meaningful noun with the verb, such as PrintFile.
- Variables and constants cannot contain a verb.
- Macros can have function names if they simulate functions.
- One-letter variables are only allowed locally as loop counters or indices (e.g. i, k or n)
- Avoid using cryptic abbreviations that could be misunderstood or lead to confusion.
- Use either only English or only German names in a source code (recommendation: English names, as these are shorter and more precise).

General formatting and coding requirements

- Block comments / *… * / are only permitted in the header of the source code and, if necessary, before function definitions. Otherwise, include all comments as line comments with //!
- If possible, use only ASCII characters in the source code and no special characters (German umlauts etc.) so that the source code is as easily portable as possible.
- Do not use the comma operator as a separator for a sequence of statements. Above all, implement declarations of variables always in a single line only, so do not use "int a, b, c, d;" (the variable names in this example are also unsuitable).

- Include comments on declarations of variables and constants (with line comment) and format the comments in the same column (one below the other). Example:

```
int i;                                  // Run variable
float temperature;                      // Variable for current temperature
char street[] = „Duisburger Str. 100“   // String array for the address
```

- Floating point numbers should always be specified with a decimal point or with an exponent:
  CORRECT: *float temperature = 20.0;* / INCORRECT*: float temperature = 20;*
- If possible, constants/markers should always be used for fixed values/expressions, as this enables very simple program adaptation!
- Use as few return statements as possible. If possible, only use the return statement at the end of the function. A return statement may only be used at the beginning after a plausibility check, if invalid values were passed to the function.
- Use the break and continue statements as rarely as possible and document them well so that it is clear which loop (in the case of interconnected loops) is aborted or whether a case/ switch selection is ended.
- Never leave out the default part in switch statements and return a default value or an error message if necessary.
- Do not omit the else part of nested “else if” statements.
- Indent by one tab per nesting level.
- The source code is to be indented exactly one more tab after an opening curly bracket. It must be indented exactly one tab less before a closing curly bracket. Tabs shall only appear at the beginning of the line and directly one behind the other. Tabs must not be mixed with spaces.
- The curly brackets should always be in the same line after the control statement. The closed curly bracket is in the column in which the associated control instruction begins. In this way, the control flow blocks can be clearly distinguished between each other. Examples:

```
for (…) {                   do {
    …           or              …
}                           } while (CONDITION)
```

- Even if there is only one statement after for, if etc., curly brackets must be used to enclose this single statement.
- The case statements and the default statement belonging to a switch are to be indented. After a case or default statement it is also necessary to indent. The default statement shall be positioned at the end (after all case statements). Example:

```
switch (EXPRESSION) {
   case x:
       ...
       break;
    ...
    default:
       ...
}
```

- Put a space between for, while, if, switch and the following bracket (since this is not a function call), example:
  CORRECT: *for (i = 0; i < argv; ++i)* / INCORRECT: *for(i = 0; i < argv; ++i)*
- Put a space around each of the assignments = += -= *= /= %= &= |= ^= <<= >>=, the operators + - * / % & | ^ && || << >> and the comparisons < <= > >= == !=, example:
  CORRECT: *a = b + c* / INCORRECT: *a=b+c*

- Do not put spaces after opening and before closing round or square brackets, example: CORRECT: *(a - b)* / INCORRECT: *(a-b)*
- Connect the operators * & [] . -> () ! ~ ++ -- to the associated variable/function with no space between them, e.g. *cp or i++
- Expressions and assignments should be kept as simple as possible. There should be no multiple side effects (such as post increment i++) in one expression, this must remain traceable and it must be clearly defined when the side effect will occur. The order of evaluation must be observed for each expression. If an expression is too complex, the expression should be simplified and, if necessary, spread over several lines, so that it can be easily understood.
- Constants should always be defined with const and not as a macro with #define.

Functions

- In the case of function prototypes, do not specify the data type only but also a meaningful parameter name for each parameter.
- In the case of function declarations/definitions, do not put a space between the function name and the opening brackets for the parameter list.
- Declare functions without return parameters as follows, otherwise int is automatically set as the return type:

  void Function(…);
- Declare functions without arguments as follows, otherwise any argument list is automatically allowed:

  … Function(void);
- Include a line break after a comma for long parameter or argument lists or if/while conditions and indent up to the bracket (using tabs + spaces!):

```
int Function(int argument1, int argument2, int argument3, int argument4,
             int argument5, int argument6)
```

- Before the definition of a function, include a (block) comment in order to explain the passed parameters and the return parameter; if necessary, also explain the function itself in brief (for more complex functions)

4. History

Version	Modification	Date	Autor
0.1	Creation of a document	14.04.2020	Schepers
0.2	Insert content in Chapters 2 and 3	27.04.2020	Schepers
0.3	Section 3: Mandatory requirement for braces added (even with only a single statement). Recommendation added: Better to use constants than macros with #define. Structure for source texts and header files formulated separately.	14.05.2020	Schepers
0.4	Correction of the English translation	01.04.2021	Schepers
0.5	Safe Temperature Monitoring	25.06.2023	Kaushik