C/C++ Coding Guideline

Revision: 1.0

Date: 01-Oct-2021

Volansys Technologies Pvt. Ltd.
Block A 7th Floor, Safal Profitaire, Corporate Road,
Prahaladnagar, Ahmedabad 380015, India
www.volansys.com
Phone: 91-79-4004-1994
Email: info@volansys.com

Confidentiality Notice

TABLE OF CONTENTS

1. INTRODUCTION

This document defines the guideline to follow common coding conventions across the organization for embedded projects having focus on programming language as C & C++ language.

1.1 Purpose

- Cover detailed guideline for the Volansys coding standard.
- Improve the **Readability** (changes and reviews). Improve the readability in repositories (e.g., Tab filled with space).
- Improve **Reliability** (code complexity, LOC, error counts). Improve **Usability** (support the document generator tools like Doxygen to extract comments from source code which is useful for API documentation).
- Improve the **testability** by above measures.

1.2 Scope

- To cover coding guideline for C and C++ language only.
- It is applicable for RTOS, Bare-metal and Linux Kernel/User-space development.
- To follow the standard by all embedded Software programmers.
- To be followed only when client/customer has not imposed their own coding standard for development.

1.3 Acronyms & Abbreviations

Terms	Definition
Bare-metal	A program written without using RTOS
IDE	Integrated Development Environment
LOC	Line of Code
K&R	Kernighan & Ritchie style (style used by C language designers)
ADT	Abstract Data Type

1.4 References

References	Provided by	Remarks
The C programming language by K&R		The text book
https://en.wikipedia.org/wiki/Cyclomatic_complexity		

2. MANDATORY RULES

2.1 Consistent indentation = 4 characters (preferred), or 2 characters. [Configure IDE, compare tool]

2.2 Disable tab keys. Tab filled with 4 spaces (preferred), or 2 spaces. [Configure IDE, compare tool]

2.3 Use K&R style braces:

```
while(!_exit)
{
    Uint8_t axisHeight_mm = 1, testCnt = 0;

    input();
    process();
    output();
}
```

2.4 Before each function, use Doxygen style comment header in function declaration present in header file (not required in source file):

```
#define CONVERT_RATIO(in,out)    (out=(pi*X_VALUE/Y_COEF))

/*************************************************************
* @brief   <function description>
* @param   <details of parameters passed to this function>
* @return  <returning parameters/values of this function>
*
*************************************************************/
void Test_process(void * paraPtr)
{
     Uint16_t temperature;

     ...
     InvokeSysMtd();            // invoke system memory technology devices
}
```

2.5 Use parentheses for each mathematic expression, logical expression, macro definition etc. (Do not rely on precedence rules of compiler/pre-processor). E.g. `Hz_555IC = 1.44/((R1+(2*R2))*C)`

2.6 Keep always a single empty line between variables' definition and procedural statements.

2.7 Use name starting with "underscore" to represent a private for variable/function/class. (You may also prefix "static" to such function or variable). The scope of such function/variable/class is within the specific module only. In C++, the underscore is represented as minus sign "-" which itself can be meaned as "private" (class or method or variable). So, the private variable or method in C++ class will prefix underscore.

2.8 Use starting "lower case" character to represent all local (auto) variables in the scope of function (or between two braces) only. All argument declarations should also start with lower case. (In example **2.4** above, see void * paraPtr).

2.9 Use function name **starting with "capital case" character to represent a public or global** function or class. The scope of this function is global and visible to other modules so that they can call it.

2.10 Use all capital letters for global-constants, global #defines & global enum. (You may use underscore as word separator). Local-constants within function may also be defined as non-capital (lower cases like `const float coil3_XL = 35.572; // ohm`).
Note: Do not prefix underscore for constant names as leading underscore names are used by C

2.11 Reduce all compiler warnings to minimum in a project. The critical warning must be resolved. Also resolve the warning generated/shown by IDE if it is valid.

2.12 Use eclipse or similar IDE for C and C++ projects. (It is not in scope of this document to explain the features of IDEs. But those features are very much useful now a days to ease and speed up the development activity).

2.13 Use typedef to specify user-data-type or ADT with this naming style:

```
typedef uint8_t      state_t;
typedef struct {…}   student_t;      // "_t" suffix to show as typedef or ADT
```

2.14 Never define data structure, variables or functions in header file. If required to export to another module, use extern keyword like: `extern float SystemVmax;` (Avoid the practise to use *extern functions*. This can be achieved easily if you comply advisory rule **3.8**).

2.15 Ensure that all variables (and pointers) are initialized before they start to use in process. Pointer can be initialized with NULL.

2.16 Use always named constants in code. In other words, avoid hard-coded numerical values without comments. For example,

```
const float PI          = 3.14;
uint32_t secondsPerDay = (60*60*24);
port0_pin3_mode     = 0x11; // input, as interrupt, -ve edge, pull-down enabled
```

2.17 If there is an equality "==" comparison against a constant, constant to be used on left side always:

a) Correct & recommended
b) If single "=" instead of "==" by mistake,
Compiler will give error.

```
const uint8_t FEMALE = 0, MALE = 1;

if (MALE == student[rollNum].gender)
{
    …
}
```

a) Correct but **not recommended**
b) If single "=" instead of "==" by mistake,
gender will modify & no compiler error!

```
const uint8_t FEMALE = 0, MALE = 1;

if (student[rollNum].gender == MALE)
{
    …
}
```

3. OTHER (ADVISORY) RULES

3.1 Naming of each variable should be meaningful and easily understood. For example, `myVar, temp1, flags` are not meaningful.
The better readable names are `CelsiusVal, motorRunFlag, _hardwareType` etc.

3.2 Function name will normally be prefixed with some action verb, e.g.: DetectHwType(), GetTemperature(), setTemperature(), _doProcess() etc. (i.e. MotorStatus() is not right name for function but suitable for variable name).

3.3 Prefer to avoid declare & define new local (auto) variable(s) randomly in a function or in any parenthesis code blocks. In other words, declare & define them in the beginning of function only.

3.4 It is preferred to avoid the abbreviations in the naming of functions and variables. If required, put a comment (near to variable defined) describing abbreviated name. For example:

```
TORQUE ProcMachinePIDPara();     // Process Machine Proportional, Integral
                                 // and Derivative Parameters to find torque
```

3.5 Avoid using global variables. If required, use through APIs from relevant module / class. If global variables defined, use prefix 'g' like `uint8_t gSystemCurrentPowerMode;`

3.6 **A)** Prefer not to use dynamic memory allocation and de-allocation in your implementation unnecessarily.
OR
B) If used, prefer to have allocation/de-allocation frequency as minimum as possible.
C) Use fixed memory (static memory) one-time allocation, if available and possible.

3.7 Avoid using "**goto**" keyword. It is also good to avoid **"continue"** keyword.

3.8 Prefer to make each source file (module) successfully compiled individually with no errors.

3.9 Prefer to use "`no optimization`" option for compiler unless specified anywhere.

3.10 Avoid using complex and unnecessary bit manipulators and bit comparisons statements.

3.11 In a module (a C file) or a class, the member variables/methods should be in this order: private, protected and public in the file.

3.12 The C++ set of casting operators static_cast, reinterpret_cast, const_cast and dynamic_cast should be used instead of C-style casting.

3.13 Each function should have maximum 7 passing arguments.

3.14 Prefer to create a function instead of a complex macro or function-like macros.

3.15 It is good to avoid unnecessary pointers and pointer arithmetic.

3.16 Avoid a same repetitive computation to derive a result. Better to save in a ram variable and use it when required. This will save execution time & reduce code space. For example,

Repetitive computation	Onetime computation

`SendToUSBChannel(pi*r*r);`
`printOnConsole("AREA = %d", pi*r*r);`
`SaveLog(epoch, r, pi*r*r);`
	`Area = pi * r * r;`
	`SendToUSBChannel(area);`
	`printOnConsole("AREA = %d", area);`
	`SaveLog(epoch, r, area);`

3.17 In some cases where repetitive computation is required then you **must have to use prefix `volatile`** to the defined variables. The keyword 'volatile' prevents compiler to perform internal optimizations which actually fails intended operation expected by programmer. See two examples below:-

Failed repetitive computation	Successful Repetitive computation

`bool clock_in, data_in;`
 	
`set_d_flipflop_output(bool level)`
`data_in = level;`
`clock_in = LOW;  // ff outpin = level`
`clock_in = HIGH; // ff outpin latched`
	`volatile bool clock_in, data_in;`
	
	`set_d_flipflop_output(bool level)`
	`data_in = level;`
	`clock_in = LOW;  // ff outpin = level`
	`clock_in = HIGH; // ff outpin latched`

• *Compiler may discard 'clock_in = LOW' considering that it has no use as next change is for 'HIGH'!* • *Compiler may omit 'data_in = level" statement as no one using'data_in' variable!* • *Compiler may discard whole this function!*	• *Compiler will take everything as programmer expects and don't do any self-optimizations.*

3.18 Keep the number of line of code per function minimum. LOC per function should not exceed 30 lines. Smaller functions are better to test, review & maintain.

3.19 **[IMPORTANT]** Keep your **code complexity level below 6** for each written function. This will benefit a maintainable, reviewable, reliable & testable code foundation. Your bug fixing turnaround time will drastically reduce!

- The benchmarking is for code complexity level derived:-

Level	Implement & review capability	Maintainability	Testability	Acceptable?
1-5	Very easy	Better	Better	Yes
6-10	Easy to average	Good	Good	Yes
11-15	Average to some difficult	Average	Poor	Sometimes
20+	Difficult to very difficult, Refactoring required	Poor	Very poor	No

- Refer various tutorials and whitepapers on "Cyclomatic Code Complexity" for further details. (Reference: https://en.wikipedia.org/wiki/Cyclomatic_complexity)
- Refer various online & offline "Cyclomatic Code Complexity calculators". Just paste your one function and see the complexity quickly. (**NOTE: Be very careful for proprietary copyrighted & licensed source code not to paste at all at online calculators. Rather use offline downloaded tools for such private source codes**).

4. File Names & Organization

4.1 Header file name extension (for both C and C++ source files) : `printer.h`

4.2 Source files name extension: `printer.c` for C, `printer.cpp` for C++.

4.3 Each source/header file must be named for single sub-system or feature.

4.4 Multiple files for a single sub-system or feature can be put into a single folder.

5. File Contents

5.1 Each file must contain only one functional system, sub-system, module or feature.

5.2 The module may have one or multiple classes / functions / processes, but must be relevant to that specific functionalities only as stated in point **5.1**. This is called **"strong cohesiveness of a system"**.

5.3 Each system must ideally be less inter-dependent, less inter-linked and less-communicated with another systems. This is called **"loose coupling among systems"**.

5.4 The example header file with its contents is given in topic **6**.

6. HEADER FILE EXAMPLE

```
#ifndef ELEVATOR_SIM_H
#define ELEVATOR_SIM_H
/*****************************************************************************************
* Elevator Simulator Application
*
* Copyright (C) 2021  Volansys Technologies Pvt Ltd
*
* Author: Ramesh Kumar, Ramesh.kumar@volansys.com
*
* Version 1.1: Fixed the issue to reduce the latency among simultaneous request within a second (05/Jan/2021)
* Version 1.0: Initial version (18/Dec/2021)
*
* This program is a licensed software. You cannot redistribute and/or modify without prior permission.
*
* This is simulator helps to simulate the elevator system & demonstrate the performance characteristics.
* other details …
*****************************************************************************************/

#include <cpu.h>
#include <math.h>

/*-----------------------------------------------------------------------------------------------------------------------*/
 *  CONSTANTS & DEFINATIONS
/*-----------------------------------------------------------------------------------------------------------------------*/

#define SECURE_TRUE     0xAA
#define SECURE_FALSE    0X55

#define FLOORS_MAX      5
#define TRAVEL_TIME_MIN 10      // 10 cm per minute

enum ELEVATOR_STATES
{
    MAINT = 0,
    STOP,
    WAIT_FOR_INPUTS,
    DOOR_OPENED,
    DOOR_CLOSED,
    RUN,
    EMERGENCY,
    STATES_MAX

} elState_t;

/*-----------------------------------------------------------------------------------------------------------------------*/
 *  DATA STRUCTURE & MACRO DECLARATIONS
/*-----------------------------------------------------------------------------------------------------------------------*/

/**
* @brief    data structure for elevator configuration by contractor
*
* @param  Floors available to process
* @param …
*/
```

```
typedef struct ConfigPara
{
    uint8_t Floors;
    uint8_t RunTimeCmPerMinute;
    uint8_t InputTimeMax;
    uint8_t DoorOpenTimeMax;
    …
} cfgPara_t;

#define MY_AREA(radius) (PI*radius*radius)
/*-----------------------------------------------------------------------------------------------------------------------*/
 * FUNCTIONS DECLARATION
/*-----------------------------------------------------------------------------------------------------------------------*/

/**
* @brief     get the elevator state
* @param  none
* @return  elState : any one state among elState_t defined states
*/
elState_t GetElevatorState(void);

/**
* @brief     set the elevator configuration parameters
* @param  paraPtr : pointer to an object filled with all the parameters as per cfgPara_t
* @return  status : success (1) or fail(0)
*/
bool SetElevatorConfig(cfgPara_t * paraPtr);


#endif // ELEVATOR_SIM_H
```

Note: The macro `#ifndef ELEVATOR_SIM_H` will proceed to include this file itself in the project having multiple source files and some of them need this header file. When this header file is first time included by some source file, this macro is created as `#define ELEVATOR_SIM_H` to avoid next inclusions by rest of source files. This is called “Header guard”.

7. REVISION HISTORY

Revision	Date	Details of change	Author	Reviewer	Approver
1.0	01-Oct-21	Initial Release	Pratik Agrawal, Bhushan Gajjar	Amit Solanki, Harshal Patel, Neha Patel, Palak Patel	Rinkal Shah