安全软件开发规范

修订记录 Revision History

版本	日期	修改内容	拟稿	审核	批准
V0.0.1	2018.7.5	创建	刘煜海		

目录

文档目的及范围

为保证项目的安全软件开发工作中，软件具有高的代码质量、良好且统一的代码风格，特编制此软件编码规范。

本文的预期读者是使用 C 语言进行安全软件相关开发工作的设计者、程序员以及软件测试人员。

本文件适用于　　控制软件。

参考资料

表 1 参考资料

序号	文件号	文件名	版本
1	N/A	MISRA-C 2004	2004
2	N/A	EN50128	2011

术语和缩写词

表 2 术语与缩写词

术语/缩写词	全称和解释
ATO	列车自动运行（Automatic train operation）
ATP	列车自动防护（Automatic train protection）
ATS	列车自动监督（Automatic Train Supervision ）
TWC	车地通信（Train Wayside Communication）
HMI	人机接口（Human Machine Interface）

总则

规范编程风格

规范编程风格，增加代码的可读性、可维护性和可测试性。

防御性代码的使用

防御性代码指在程序处理变量之前为了保证该变量合法而增加的代码。

在使用以下代码时必须增加防护代码：

- 函数参数（防止非法参数）
- 指针（防止指针越界或为空）
- 数组（防止数组越界）
- 减法运算（防止出现小数减大数）
- 除法运算（防止除 0 操作等）

禁止使用 C 语言灵活的特性

禁止使用 C 语言灵活的特性：

- 禁止使用++，--操作符（循环控制和单语句除外）
- 禁止使用+=，*=，&=等操作
- 在变量运算时，必须使用括号将各级运算分割开，禁止使用 C 语言本身操作符的优先级表示。

宏定义的使用

- 限制使用宏嵌套宏；
- 限制使用宏来代替函数或语句。
- 允许在调试中使用宏编译，简化调试工作。

功能模块划分

程序各个功能模块应做到功能单一。

指针的使用

禁止使用三重及以上指针。

结构体的使用

允许结构体嵌套，层数不超过 3 层。

链表的使用

禁止使用动态链表。

禁止使用递归函数

循环的使用

- 循环嵌套禁止超过三层。
- 循环体内必须包含有效地循环结束条件。（任务函数与 MAIN 函数除外）

有符号数的使用

- 在有符号数定义时明确标示出其使用位置及使用原因。
- 在使用有符号数位置必须说明。（注释中表示）
- 有符号数计算时必须采用强制类型转换

函数规模

单一函数代码量应限制在 200 行以内（不包含注释）。（任务函数与 MAIN 函数除外）

浮点数的使用

尽量避免使用浮点数。

编译器报警的处理

须对编译器所报的错误和报警信息逐一排查，错误条目必须消除，对于不能消除报警的条目应具备充分的理由，不能消除的报警及其被保留的理由须被记录。

汇编语句的使用

使用汇编语言时，须进行封装和独立，在使用汇编语言时，须说明使用汇编语言的原因。

包含文件的使用

禁止在代码中直接包含“.c”文件。

扇入扇出指标

各模块扇出值均应小于等于 10，如大于规定值需进行必要的说明并与软件测试人员进行协商解决。

扇入值不做具体要求。

圈复杂度指标

函数的圈复杂度数值应小于等于 20，如大于规定值需进行必要的说明并与软件测试人员进行协商解决。

编码风格

文件结构

向工程中添加新文件时，一般分为两个文件，一个文件为头文件，其后缀为“.h”，另一个文件保存程序的实现为定义文件，其后缀为“.c”。对于功能模块来说，同时应增加 ver_module.c 文件来维护模块版本号等对模块的描述性内容。

文件的声明

在每一个头文件以及定义文件中必须包含文件的声明（每个文件只能有一个声明）。

定义文件的结构

定义文件有三部分内容：

- 定义文件开头处的版权和版本声明。
- 对一些头文件的引用。
- 程序的实现体（包括数据和代码）。

以上三个部分必须按照上述顺序排列。

文件信息说明

文件必须添加文件头信息，文件头信息应至少包含以下信息：

- 文件名，公司信息
- 文件简要描述
- 创建者
- 创建时间
- SVN 记录
- 修改记录(功能模块在版本控制文件中维护，其他在文件中描述)

示例：

/**

- @file ver_abc.c
- @brief abc 模块版本描述文件

- @author
- @date 2010-11-03 *
- $Id: ver_abc.c 35 2012-05-03 00:39:27Z $
- @history:

1. Date:

Author:

Modification:

2. ....

*/

函数

函数说明

在每一个函数中必须包含函数的说明（每个函数只能有一个说明）。函数说明应至少包含以下信息：

- 函数简要描述
- 输入参参数
- 返回值，应增加说明返回值可能的范围及含义

示例：

/** *

- @brief get_input 向外部提供 IO 数据
- @param type IO 类型
- @param buff IO 数据缓存
- @return IO 数据大小

*/

函数的命名

函数的命名宜采用动宾结构命名，函数名称应能最大程度表述出函数的功能。应避免无意义的函数命名和容易产生歧义的函数命名，如：拼音等方式。

函数的参数

- 函数的参数数量不宜超过 6 个。

- 函数的参数的命名要意义明确，尽量避免与全局变量重名。
- 无参数的函数必须填充 void 关键字。

函数返回值

函数内有且只有一个 return 语句。

函数声明

- 单文件(.c 文件)中使用的函数，需要在文件顶端增加 static 函数声明。
- 多文件(.c 文件)中使用的函数，需要在单独的.h 文件中增加函数声明。其中，提供给外部模块或文件使用的，在.h 文件中声明时不加 extern；本模块或文件调用外部模块的函数，在.h 中声明时增加 extern。

宏定义与枚举变量

宏定义和枚举变量命名方式

宏定义和枚举变量应使用大写字母和下划线的组合命名。其中，宏定义应使用动宾结构短语或者形容加名词的形式来命名。枚举变量使用 TYPENAME_TYPE 的方式来命名,枚举变量赋值有两种方式：

- 采用默认值
- 手动赋值(所有变量都需要手动赋值，严格避免只赋值其中的几个变量)

当需要使用大量宏定义定义类型时，宜采用枚举变量类型来替代。

如，应使用

enum{

BALIAS_UP_MAIN = 0, /-< *上行主信号应答器*/

BALIAS_DOWN_MAIN = 1, /-< *下行主信号应答器*/

BALIAS_UP_PRE = 2, /-< *上行预告应答器*/

BALIAS_DOWN_PRE = 3, /-< *下行预告应答器*/

BALIAS_NORMAL = 4, /-< *普通应答器*/

};

来替代

```
#define BALIAS_UP_MAIN  (0)      /*-< 上行主信号应答器*/

#define BALIAS_DOWN_MAIN (1)        /*-< 下行主信号应答器*/
```

```
#define BALIAS_UP_PRE (2)            /*-< 上行预告应答器*/

#define BALIAS_DOWN_PRE (3)      /*-< 下行预告应答器*/

#define BALIAS_NORMAL (4)            /*-< 普通应答器*/。
```

宏定义的使用

- 单文件(.c 文件)中使用的宏，直接定义在本文件顶端(头文件包含段和代码段之间)。
- 多文件(.c 文件)中使用的宏，需要单独定义在.h 文件中。

变量

变量的命名

- 变量的命名推荐采用采用 Unix 风格（即小写加下划线命名），部分与外界接口可按照相关接口处理。
- 禁止使用简单无意义的变量命名，例如 i，j，k 等。（循环控制等除外）
- 静态变量需要增加前缀“s_”，全局变量需增加前缀“g_”或“_”。
- 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。
- 局部变量不得使用和全局变量相同的名字，防止作用域的混淆。

变量的初始化

- 变量在使用前必须进行初始化操作。
- 全局变量必须在系统上电后对其初始化操作。
- 局部变量必须在定义后进行初始化操作。

变量的类型转换

变量类型转换须采用显式方式进行。

全局变量的定义及声明

- 单文件域(.c 文件)中使用的全局变量，需要用 static 关键字定义在本文件(.c 文件)顶端。
- 多文件(.c 文件)中使用的全局变量，需要单独定义在.c 文件中，同时在单独的.h 文件中增加声明。

表达式与基本语句

If..else 推荐编码结构

/- 标准注释 0: 不同行注释/

```
if (((a == 0) && (b == 0)) || (c == 0)) /-< 标准注释 1: 同行注释/
{
do_something;
}
else if (d != 0) /-< 标准注释 2: 同行注释/
{
do_something;
}
else
{
;
}
```

For、while、do..while 推荐编码结构

```
for (i = 0U; i < 10U; i++)
{
do_something;
}
while (a != 0U)
{
do_something;
}
do
{
do_something;
}
while (a != 0)
```

Switch 推荐编码结构

```
switch (number)
{
case 1U:
case 2U:
break;
case 3U:
break;
default:
break;}
```

函数推荐编码结构

```
d = function (a, b, c);
```

注释

提高可读性

在部分关键代码（现场调试及关键逻辑代码），容易混淆的定义或者声明，或者其他程序员觉得有必要添加注释的地方，需添加清晰，简要的注释。在某些可能同时有较多人员进行修改的文件中，注释要尽量丰富，以下列出部分必须注释的条目：

- 数学常量
- 关键变量及全局变量
- volatile 型变量应给出可能改变该变量的函数或代码
- 信号量应给出等待和触发的函数或代码

注释类型

注释推荐如下风格：

- 函数内同行内容注释：/-< /
- 函数内不同行内容注释：/- /
- 函数外内容注释：/* */

注释风格

- 一般情况下，源程序有效注释量必须在 20％以上。

- 注释应与其描述的代码相近，对代码的注释应放在其上方或右方相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

示例：

应如下书写：

/- *注释示例* /

repssn_ind = ssn_data[index].repssn_index; /-< *注释示例* /

repssn_ni = ssn_data[index].ni;

- 对于所有具有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：

/- *宏定义说明*/

```
#define MAX_ACT_TASK_NUMBER 1000 /*-< 宏定义说明*/
```

- 注释与所描述内容进行同样的缩排。

示例：

void example_fun(void)

{

/- *code one comments* /

Code Block One

/- *code two comments* /

Code Block Two

}

- 将注释与其上面的代码用空行隔开。

示例：

应书写为如下形式：

/* code one comments */

program code one

/* code two comments */

program code two

自文档化

为了充分利用注释，在开发过程中应将注释内容和文档相结合。对于可能在文件外部引用的函数、变量和宏，一定要按照 doxygen 和 cdocgen 文档工具的格式进行注释，以方便生成接口描述文档。

关于 doxygen 文档工具的使用格式和方法，参见 doxygen 手册。

cdocgen 目前使用/** /*风格的 doxygen 注释和/- /*来提取相关的信息，推荐在函数头部使用 doxygen 风格，在函数体中使用/- */，以便和 doxygen 达到更好地可兼容性。代码段的标记用于进行流程图的绘制。

代码段的注释中,/** /*和/- */注释默认将用于下一行,/**< /*和/-< */默认将注释用于当前行。

doxygen 的注释中还支持@ 作为关键字的前缀字符。典型的函数头部注释包括：

@brief 为函数的概述

@param 为函数的参数

@return 为函数的返回值

@detail 函数详细说明

@depth 为函数的深度

@alias 可以强制将某个组合语句合并为一个节点

@details 可以补充详细说明

@ignore 将跳过本函数的详细设计

预处理指令

为规范预处理指令的使用，以下给出几种常用的预处理指令用法。

防止重复引用

在头文件中，应在文件头部和尾部使用类似

```
#ifndef  _DEBUG_H

#define  _DEBUG_H

……
```

```
#endif   /*-< _DEBUG_H */
```

的结构来防止重复引用。

配置选项开关/增强可移植性

在某些模块中，可引入一个宏定义作为选项开关，并在代码中通过宏定义的不同数值来决定执行不同的策略。如：

```
#define ENABLE_NEW_ALGORITHM (1)

……

#if ENABLE_NEW_ALGORITHM == 1

do_sth();

#else

do_other_thing()

#endif
```

配合配置选项开关，宏定义还可用于重新定义某语句在不同平台或者不同选项下的效果，从而增加程序代码的可移植性。如：

```
#ifdef DEBUG_PORT==1

#define send(x) uart1_send(x)

#elif DEBUG_PORT==2

#define send(x) uart2_send(x)

#endif
```

调试打印信息

C 语言提供了诸如__FILE__，**LINE**，**DATE**，__TIME__的系统宏来提供文件或者语句的相关信息，应善用这些由预处理器进行自动填充的宏。

如，在版本字符串中使用：

```
#define VERSION_STR "\r\n 自动驾驶系统(ATO),CRSCD\r\n""版本"BRANCH_STR
ATO_VERSION"\r\n""编译时间: "__DATE__" " __TIME__"\r\n"
```

来定义 ATO 的版本号，则 VERSION_STR 会被自动展开，并将本文件的编译时间嵌入到 VERSION_STR。

又如，在进行分支路径的判断时，可使用__LINE__展开的方式来确定所执行语句在源代码中的行数：

```
#define TRACE_CMD printf("CMD at Line %d\n", __LINE__)
```

……

```
if (state == STATE_UP)
{
judge_cmd();
TRACE_CMD;
}
Else
{
check_cmd();
TRACE_CMD;
}
```

从而，当程序执行到该分支时，由于两处的 TRACE_CMD 中，关键字__LINE__将被扩展成宏定义调用时的行数，从而可在调试的时候反查处程序流的执行情况。在该模块的调试工作完成之后，可以将 TRACE_CMD 宏定义为空，从而在正式发布版本中将调试代码全部去掉。

规则

规则引用

本文档遵循《MISRAC-2004 编码规范》中所有规则。

补充规则

补充规则 1.1(强制)：禁止使用浮点数。

MISRA-C：2004 编码规范对应表

修改规则项	不采纳或修改理由	规范内容
12.1	建议改为强制	不要过分依赖 C 表达式中的运算符优先规则

/