目录

深圳信盈达 C 程序编程规范

1.1 数据类型定义

进行任何软件编程前，第一步要做的就是先重新定义数据类型名，这样方便移植，现在的很多软件都已经包含内置的数据类型重定义的头文件。以 Keil MDK 软件以为，其他标准定义文件是 stdint.h。 以上是其部分代码片段，使用 KEIL 开发编程建议直接包包含 stdint.h ，如果使用的开发环境没有重新定义以下数据类型名，请复制以下文件保存成 mystdint.h，加入自己的工程头文件目录中。

```
/* Copyright (C) ARM Ltd., 1999 */
/* All rights reserved */

/*
 * RCS $Revision: 178085 $
 * Checkin $Date: 2012-12-11 14:54:17 +0000 (Tue, 11 Dec 2012) $
 * Revising $Author: agrant $
 */

#ifndef __stdint_h
#define __stdint_h

/*
 * 'signed' is redundant below, except for 'signed char' and if
 * the typedef is used to declare a bitfield.
 * '__int64' is used instead of 'long long' so that this header
 * can be used in --strict mode.
 */

    /* 7.18.1.1 */
    /* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short    int int16_t;
typedef   signed          int int32_t;
typedef   signed          __int64 int64_t;

    /* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short    int uint16_t;
typedef unsigned          int uint32_t;
typedef unsigned          __int64 uint64_t;

    /* 7.18.1.2 */

    /* smallest type of at least n bits */
    /* minimum-width signed integer types */
typedef   signed          char int_least8_t;
typedef   signed short    int int_least16_t;
typedef   signed          int int_least32_t;
typedef   signed          __int64 int_least64_t;
```

```
    /* minimum-width unsigned integer types */
typedef unsigned           char uint_least8_t;
typedef unsigned short     int uint_least16_t;
typedef unsigned           int uint_least32_t;
typedef unsigned           __int64 uint_least64_t;

    /* 7.18.1.3 */

    /* fastest minimum-width signed integer types */
typedef   signed           int int_fast8_t;
typedef   signed           int int_fast16_t;
typedef   signed           int int_fast32_t;
typedef   signed           __int64 int_fast64_t;

    /* fastest minimum-width unsigned integer types */
typedef unsigned           int uint_fast8_t;
typedef unsigned           int uint_fast16_t;
typedef unsigned           int uint_fast32_t;
typedef unsigned           __int64 uint_fast64_t;

    /* 7.18.1.4 integer types capable of holding object pointers */
typedef   signed           int intptr_t;
typedef unsigned           int uintptr_t;

    /* 7.18.1.5 greatest-width integer types */
typedef   signed           __int64 intmax_t;
typedef unsigned           __int64 uintmax_t;

#endif /* __stdint_h */

/* end of stdint.h */
```

头文件通用模板：

```
#ifndef __MYDEF_H__
#define __MYDEF_H__

    //你的内容

#endif
```

说明：上面 MYDEF_H 对应于文件名 mydef.h 。

注意：头文件中不得定义变量！

1.2 代码排版

1.2.1 级间缩进

程序块要采用缩进风格编写，缩进的空格数为 **4** 个。

说明：对于由开发工具自动生成的代码可以有不一致。

函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格， **case** 语句下的情况处理语句也要遵从语句缩进要求。

结构体定义示例：

```
struct student
{
    char name[50];
    char age;
    char address[200]
}
```

函数示例：

```
void fun_test(int value)
{
    printf("%d\r\n",value);
}
```

判断语句示例：

```
int max(int a,int b)
{
    if (a > b)
    {
        return a;
    }

    else
    {
        return b;
    }
}
```

switch 语句示例：

```
switch (cmd)
{
    case LED_ON:
        ....;
        break;

    case LED_OFF:
        ....;
        break;
    default:
        break;
```

```
}
```

1.2.2 代码块空间规则

相对独立的程序块之间、变量说明之后必须加空行。

示例：如下例子不符合规范。

```
if    (!valid_ni(ni))
{
      ... // program code
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

应如下书写

```
if    (!valid_ni(ni))
{
      ... // program code
}
                                                    //注意，这里空了一行
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

1.2.3 长语句断行书写

较长的语句（ >80 字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

一行程序以小于 **80** 字符为宜，不要写得过长。

示例：

```
perm_count_msg.head.len = NO7_TO_STAT_PERM_COUNT_LEN
                                + STAT_SIZE_PER_FRAM * sizeof( _UL );

act_task_table[frame_id * STAT_TASK_CHECK_NUMBER + index].occupied
                          = stat_poi[index].occupied;

act_task_table[taskno].duration_true_or_false
                          = SYS_get_sccp_statistic_state( stat_item );

report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
                          && (n7stat_stat_item_valid (stat_item))
                          && (act_task_table[taskno].result_data != 0));
```

1.2.4 短语句书写

不允许把多个短语句写在一行中，即一行只写一条语句。

示例：如下例子不符合规范。

```
rect.length = 0; rect.width = 0;
```

应如下书写

```
rect.length = 0;
rect.width = 0;
```

1.2.5 流程控制语句块规则

1. **if**、 **for**、 **do**、 **while**、 **case**、 **switch**、 **default** 等语句自占一行，且 **if**、 **for**、 **do**、 **while** 等语句的执行语句部分无论多少都要加括号**{}**。

示例：如下例子不符合规范。

```
if (pUserCR == NULL) return;
```

应如下书写：

```
if (pUserCR == NULL)
{
    return;
}
```

2. 关键字之后要留空格,if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。

示例：

```
for (i = 0; I < MAX_BSC_NUM; i++)
{
    do_something(i_width, i_height);
}
```

1.2.6 tab 键，空格键使用

对齐只使用空格键，不使用 TAB 键。

说明：以免用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐，不要使用 BC 作为编辑器合版本，因为 BC 会自动将 8 个空格变为一个 TAB 键，因此使用 BC 合入的版本大多会将缩进变乱。

1.2.7 程序块的分界符

程序块的分界符（如 **C/C++**语言的大括号‘ **{**’和‘ **}**’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及 **if**、 **for**、 **do**、 **while**、 **switch**。

不规范示例：

```
int max(int a,int b)
{
    if (a > b){
        return a;
    }

    else{
        return b;
    }
}
```

```
for (...) {
    ... // program code
}

if (...)
    {
        ... // program code
    }
```

```
void example_fun( void )
    {
        ... // program code
    }
```

规范示例：

```
int max(int a,int b)
{
    if (a > b)
    {
        return a;
    }

    else
    {
        return b;
    }
}
```

```
for (...)
{
    ... // program code
}

if  (...)
{
    ... // program code
}

void example_fun( void )
{
    ... // program code
}
```

1.2.8 流程控制语句建议

1. 有可能的话， **if** 语句尽量加上 **else** 分支，对没有 **else** 分支的语句要小心对待； **switch** 语句必须有 **default** 分支。
2. 不要滥用 **goto** 语句。
   说明： goto 语句会破坏程序的结构性，所以除非确实需要，最好不使用 goto 语句。

1.2.9 运算符书写

1. 二元运算符号：对于左右两都带参数的运算符号，像+,-,*,/,<,等符号两边都需要空一个空格。
   如：a = b + c;
2. 非二元运算符：像++，--，变量和运算符号间无空格。
   如：i++, ++i;

1.3 程序注释

1.3.1 注释量规定

一般情况下，源程序有效注释量必须在 **20**%以上。

说明：注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

1.3.2 头文件注释

文件头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、修改日志等。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/*********************************************************************************
 * Copyright: 2008-2015, 信盈达科技有限公司
 * File name:   文件名
 * Description: 用于详细说明此程序文件完成的主要功能，与其他模块或函数的接口，
 *  输出值、取值范围、含义及参数间的控制、顺序、独立或依赖等关系
 * Author: 作者
 * Version: 版本
 * Date: 完成日期
 * History: 修改历史记录列表，每条修改记录应包括修改日期、修改者及修改内容简述。
*********************************************************************************/
```

1.3.3 函数注释

模板如下：

```
/*********************************************************************
*   Function：        函数名称
*   Description:      函数功能、性能等的描述;
*   Calls:            被本函数调用的函数清单
*   Called By:        调用本函数的函数清单
*   Input：           输入参数说明，包括每个参数的作输入参数说明，包括每个参数的作用;
*   Output：          对输出参数的说明;
*   Return :          函数返回值的说明;
*   Author:           作者
*   Others:           其他注意事项说明
*   date of completion:   完成日期
*   date of last modify:   最后修改日期
*********************************************************************/
```

1.3.5 关于注释建议

1. 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。
2. 注释的内容要清楚、明了，含义准确，防止注释二义性。
3. 说明：错误的注释不但无益反而有害。
4. 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（对单条语句的注释）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：如下例子不符合规范。

例 1：

```
/* get replicate sub system index and net indicator */
                                                    //这里不应该多一行
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

例 2：

```
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* get replicate sub system index and net indicator */
```

应如下书写

```
/* get replicate sub system index and net indicator */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

5. 对于所有有物理含义的变量、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：

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000
#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

6. 数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。

示例：可按如下形式说明枚举/数据/联合结构。

```
/* sccp interface with sccp user primitive message name */
enum SCCP_USER_PRIMITIVE
{
    N_UNITDATA_IND, /* sccp notify sccp user unit data come */
    N_NOTICE_IND, /* sccp notify user the No.7 network can not */
    /* transmission this message */
    N_UNITDATA_REQ, /* sccp user's unit data transmission request*/
};
```

7. 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。

示例：

```
/* The ErrorCode when SCCP translate */
/* Global Title failure, as follows */              // 变量作用、含义
/* 0 － SUCCESS 1 － GT Table error */
/* 2 － GT error Others － no use */          // 变量取值范围
/* only function SCCPTranslate() in */
/* this modual can modify it, and other */
/* module can visit it through call */
/* the function GetGTTransErrorCode() */     // 使用方法
BYTE g_GTTranErrorCode;
```

8. 注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

示例：如下例子，排版不整齐，阅读稍感不方便。

```
void example_fun( void )
{
/* code one comments */
    CodeBlock One
        /* code two comments */
    CodeBlock Two
}
```

应改为如下布局：

```
void example_fun( void )
{
    /* code one comments */
    CodeBlock One

    /* code two comments */
    CodeBlock Two
}
```

9. 避免在一行代码或表达式的中间插入注释。

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

10. 在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

示例：如下注释意义不大。

```
/* if receive_flag is TRUE */
if (receive_flag)
```

而如下的注释则给出了额外有用的信息。

```
/* if mtp receive a message from links */
if (receive_flag)
```

11. 在程序块的结束行右方加注释标记，以表明某程序块的结束。

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

```
if (...)
{
    // program code
    while (index < MAX_INDEX)
    {
        // program code
    } /* end of while (index < MAX_INDEX) */ // 指明该条 while 语句结束
} /* end of if (...)*/ // 指明是哪条 if 语句结束
```

12. 注释格式尽量统一，建议使用“ /* …… */”。

13. 注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能

用非常流利准确的英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

1.4 宏，变量，函数定义规则

关于宏，变量，函数名命名，者要遵守一些共同的规则：

1. 标识符要采用英文单词或其组合，便于记忆和阅读，切忌使用汉语拼音来命名。

标识符应当直观且可以拼读，可望文知义，避免使人产生误解。程序中的英文单词一般不要太复杂，用词应当准确。

2. 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

add /remove；begin / end；create / destroy； insert / delete ；

first/last； get / release； increment/decrement ； put / get ；

lock / unlock ； open / close ； min / max ； old / new ；

start / stop ； next / previous ； source / target ；

show / hide ； send / receive ； source / destination ；

cut / paste ； up / down

1.4.1 宏命定义

1. 宏名全部使用大写,如果变量由多个单组成，使用下划线连接

不规范写法：

```
#define     Pi      3.14159
#define     pi      3.14159
```

规范写法：

```
#define     PI      3.14159
```

2. 用宏定义表达式时，要使用完备的括号。

示例：如下定义的宏都存在一定的风险。

```
#define RECTANGLE_AREA( a, b )  a * b
#define RECTANGLE_AREA( a, b )  (a * b)
#define RECTANGLE_AREA( a, b )  (a) * (b)
```

正确的定义应为：

```
#define RECTANGLE_AREA( a, b ) ((a) * (b))
```

3. 将宏所定义的多条表达式放在大括号中。

示例：下面的语句只有宏的第一条表达式被执行。为了说明问题， for 语句的书写稍不符规范。

```
#define INTI_RECT_VALUE( a, b )\
        a = 0;\
        b = 0;
        for (index = 0; index < RECT_TOTAL_NUM; index++)
            INTI_RECT_VALUE( rect.a, rect.b );
```

正确的用法应为：

```
#define INTI_RECT_VALUE( a, b )\
                {\
                    a = 0;\
                    b = 0;\
```

```
}
for (index = 0; index < RECT_TOTAL_NUM; index++)
{
    INTI_RECT_VALUE( rect[index].a, rect[index].b );
}
```

4.　使用宏时，不允许参数发生变化。

示例：如下用法可能导致错误。

```
#define SQUARE( a )     ((a) * (a))
int a = 5;
int b;
b = SQUARE( a++ ); // 结果：  a = 7，即执行了两次增 1。
```

正确的用法是：

```
b = SQUARE( a );
a++; // 结果：  a = 6，即只执行了一次增 1。
```

1.4.2 变量命名规则

1.　基本原则：

1）变量命名全部使用小写，单词间使用“_”连接

2）变量名＝属性＋类型＋对象描述

下面是给出建议性的变量命名规范：

属性部分:

全局变量 g	结构体中成员变量 m	局部变量 l
常量 k	静态变量 s	

类型部分:

结构体 **t**	浮点型 f	整型 i
指针 p	双字 dw	枚举 em
函数 fn	字符串 sz	字节 by
无效 v	短整型 n	字 w
句柄 h	双精度浮点 d	实型 r
长整型 l	计数 cnt	无符号 u
布尔 b	字符 chr	数组 arr

描述部分:

最大 max	初始化 init	源对象 src
最小 min	临时变量 tmp	目的对象 dest

示例:

定义一个全局变量：gi_system_mode;　　→ 表示系统工作模式

定义一个局部变量：li_receive_buffer;　　→ 表示接收缓冲

变量名太长可以缩写，通常是把单词中的元音字母去除，如下：

示例：如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp ;

flag 可缩写为 flg ;
statistic 可缩写为 stat ;
increment 可缩写为 inc ;
message 可缩写为 msg ;
欢迎大家一起补充常用单词简写!!!!

2. 变量名长度应小于 31 个字符，以保持与 ANSI C 标准一致。不得取单个字符（如 i、j、k 等）作为变量名，但是局部循环变量除外。
3. 程序中局部变量不要与全局变量重名。 尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。
4. 结构名、联合名、枚举名由前缀 t 开头。

1.4.3 函数定义规则

1. 函数名全部小字字母，单词间使用 “_” 连接。
2. 函数名之后不要留空格。函数名后紧跟左括号‘(’，以与关键字区别。

   示例：

```
int max(int a,int b)
{
    …..
    return max
}
```

3. 如果一个函数功能仅提供给本模块内的函数使用，必须使用 static 修饰，对外不公开。
4. 对所调用函数的错误返回码要仔细、全面地处理。
5. 明确函数功能，精确（而不是近似）地实现函数设计。
6. 编写可重入函数时，应注意局部变量的使用（如编写 C/C++语言的可重入函数时，应使用 auto 即缺省态局部变量或寄存器变量）。

   说明：编写 C/C++语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。
7. 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即 P、 V 操作）等手段对其加以保护。

   说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

   示例：假设 Exam 是 int 型全局变量，函数 Squre_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
unsigned int example( int para )
{
    unsigned int temp;
    Exam = para; //  （ **）
    temp = Square_Exam( );
    return temp;
}
```

8. 函数的规模尽量限制在 200 行以内。

说明：不包括注释和空格行。

9. 一个函数仅完成一件功能，不要设计多用途面面俱到的函数。

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难。

10. 检查函数所有参数输入的有效性。

11. 检查函数所有非参数输入的有效性，如数据文件、公共变量等。

12. 函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入之前，应进行必要的检查。

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入之前，应进行必要的检查。

13. 使用动宾词组为执行某操作的函数命名。如果是 **OOP** 方法，可以只有动词（名词是对象本身）。

示例：参照如下方式命名函数:

```
void record_print ( unsigned int rec_ind ) ;
int record_input ( void ) ;
unsigned char color_get_current( void ) ;
```

14. 防止把没有关联的语句放到一个函数中。

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

示例：如下函数就是一种随机内聚。

```
void Init_Var( void )
{
    Rect.length = 0;
    Rect.width = 0; /*  初始化矩形的长与宽  */
    Point.x = 10;
    Point.y = 10; /*  初始化“点”的坐标  */
}
```

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

```
void Init_Rect( void )
{
    Rect.length = 0;
    Rect.width = 0; /*  初始化矩形的长与宽  */
}

void Init_Point( void )
{
    Point.x = 10;
    Point.y = 10; /*  初始化“点”的坐标  */
}
```

1.5 可读性

1. 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例：下列语句中的表达式

```
word = (high << 8) | low ;        (1)
if ((a | b) && (a & c))           (2)
    if ((a | b) < (c & d))        (3)
```

如果书写为

```
high << 8 | low
a | b && a & c
a | b < c & d
```

由于

```
high << 8 | low  等效于：  ( high << 8) | low,
a | b && a & c  等效于：    (a | b) && (a & c),
```

(1)(2)不会出错，但语句不易理解；

```
a | b < c & d = a   | ( b < c) & d,
```

(3)造成了判断条件出错。

2. 避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

示例：如下的程序可读性差。

```
if (Trunk[index].trunk_state == 0)
{
  Trunk[index].trunk_state = 1;
  ... // program code
}
```

应改为如下形式：

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1
if (Trunk[index].trunk_state == TRUNK_IDLE)
{
  Trunk[index].trunk_state = TRUNK_BUSY;
  ... // program code
}
```

3. 源程序中关系较为紧密的代码应尽可能相邻。

说明：便于程序阅读和查找。

示例：以下代码布局不太合理。

```
rect.length = 10;
char_poi = str;
rect.width = 5;
```

若按如下形式书写，可能更清晰一些。

```
rect.length = 10;
rect.width = 5; // 矩形的长与宽关系较密切，放在一起。
char_poi = str;
```

4. 不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

示例：如下表达式，考虑不周就可能出问题，也较难理解。

```
* stat_poi ++ += 1;
* ++ stat_poi += 1;
```

应分别改为如下:

```
*stat_poi += 1;
stat_poi++;          // 此二语句功能相当于“ * stat_poi ++ += 1; ”
++stat_poi;
*stat_poi += 1;      // 此二语句功能相当于“ * ++ stat_poi += 1; ”
```

说明：本手册主要参考华为，中兴公司规范，综合二者修改而成，仅限深圳信盈达公司工程师内部人员使用，不得对外传播！