

嵌入式 C 进阶之道

前言

C 语言的书有一大堆,嵌入 C 语言的书也不少,但都不过是简单介绍一下标准 C 语言的语法,再讲一下嵌入式 C 语言与标准 C 的区别,讲一下新增加的关键字。这样的书,对于初学者或许是适合的。问题是,若是我不愿意只是当一个小菜鸟,我想对嵌入式 C 语言有更多更深的了解时,我突然发现,满图书馆的书,竟找不到一本,能解我心头之惑。

对于在实际工作当中,你不再是独自编写程序,你要和小组内的成员之间分工合作,你要学会模块化编程、要写出更规范更安全的代码、做更合理的优化、减少更多的 bug。所有的这些,都迫使你必须更彻底的理解嵌入式 C 的语法结构,数据细节,与硬件打交道的特性,使得你必须时时考虑硬件与 C 的对应关系并养成良好的编程习惯。

本文的原意是想尽可能多的解决上述问题,帮助更多的新人深入理解嵌入式 C 语言。

本文是我工作之余整理而来。是对我个人学习嵌入式 C 语言过程的总结。本文涉及的知识点多数来自于网络,其中加入我个人理解以及自己平时遇到的注意点。

本文从编程风格谈起,讲述了模块化的编程方法,对一些大型项目中常用重点关键字做了讲解,参照 MISRA C 2004 规范,对嵌入式 C 安全编程做了阐述,对一些嵌入式 C 的小技巧进行了讨论。

在读本文时,我假设你有 C 语言基础、至少理解一种微控制器(51、PIC、AVR、DSP、ARM 等)、有简单的(这里本想写较深的)汇编基础。

声明:本人不拥有该文档的版权,任何人可随意传播。本人不对文中任何事物负责,(呵呵,工作比较忙,交流尽量通过邮箱),但会不定期发布新版本以改正前版的不足。希望本文能帮助一些热爱嵌入式编程的新人们,所以,如果您有更好的建议、发现本文档的错误,请将以上详细信息发到我邮箱,我会在新版本后面,注明您的名字(或者网名)。希望广大的嵌入式爱好者们能共同努力,为后来者们铺一个更平坦的大道。

一. 良好的编程风格

编程的总则：编程首要是要考虑程序的可行性，然后是可读性、可移植性、健壮性以及可测试性。大多数程序员只是关注程序的可行性，而忽略了可读性、可移植性和健壮性，其实我个人认为，程序的可行性和健壮性与程序的可读性有很大的关系，能写出可读性很好的程序的程序员，他写的程序的可行性和健壮性必然不会差，也会有不错的可移植性。程序的可读性需要程序员有一个良好的编程风格。

好风格应该成为一种习惯。如果你在开始写代码时就关心风格问题，如果你花时间去审视和改进它，你将会逐渐养成一种好的编程习惯。一旦这种习惯变成自动的东西，你的潜意识就会帮你照料许多细节问题，甚至你在工作压力下写出的代码也会更好。

1. 排版

a. 代码缩进空格数为4个。若是可能，尽量用空格来代替Tab键，因为有些编译器不支持Tab键（我自己至今未见过，但确实有这个风险），这给程序的移植带来了问题。在keil中这个问题很容易解决，只需在在keil主界面的菜单栏点击Edit—Configuration...，弹出Configuration窗口，点击Editor标签，在其中C/C++ File:、ASM、Other Files栏下，选中Insert spaces for tab: 复选框，Tab对应的框中填4，这样按tab键就相当于按下四个空格键。

```
BOOL BufClr(UINT8 * dest,UINT32 size)
```

```
{
    if(NULL ==dest || NULL==size)
    {
        return FALSE;
    }
}
```

b. 较长的语句要分2行来书写，并用‘\’符号隔开。

```
uncrc=calcCRC16(Packet.p,unlen);
if((UINT8) uncrc != Packet.down_ser.mCrc[0] \
|| (UINT8)(uncrc>>8) != Packet.down_ser.mCrc[1])
{
    BELL(ON);
}
```

c. 函数代码的参数过长，分多行来书写。

```
void UARTSendAndRecv(UINT8 *ucSendBuf,
                     UINT8 ucSendLength,
                     UINT8 *ucRecvBuf,
                     UINT8 ucRecvLength)
{
    .....
}
```

d. if、do、while、switch、for、case、default等关键字，必须加上大括号{}。

```
if(bSendEnd)
{
    BELL(ON);
}
```

```

}
else
{
    BELL(OFF);
}
//-----
for(i=0; i< ucRecvLength; i++)
{
    ucRecvBuf[i]=i;
}
//-----
switch(ucIntStatus)
{
    case USB_INT_EP2_OUT:
    {
        USBCiEP2Send(USBMainBuf, ucrecvLen);
        USBCiEP1Send(USBMainBuf, ucrecvLen);
    }
    break;
case USB_INT_EP2_IN:
    {
        USBCiWriteSingleCmd (CMD_UNLOCK_USB);
    }
    break;
    .....
}

```

2. 注释

a. 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。

注释应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。

尽量避免在注释中使用缩写，特别是不常用缩写。

注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。

b. 说明性文件必选在文件头着重说明，例如*.c、*.h文件

```

/*****
*
*                               定时器+计数器测频
*
* 文    件: frequency.c
* 作    者: 小瓶盖
* 说    明: 定时器+计数器测频率
* 编写时间: 2010.3.17
* 版    本: 1.0
* 修改日期: 无

```

*-----

* 注：本程序定义6个数码管, 经过实测, 在200HZ~50KHZ时结果较准确, 误差小于0.4%,

* 50KHZ以上频率未进行测量. 据资料表明, 可以测量到120KHZ, 本程序未证明.

*****/

```
#include <xxxx.h>
```

```
void func(void)
```

```
{
}
```

c. 函数头应该进行注释, 例如函数名称、输入参数、返回值、功能说明。

```
/******将所有参数写入 AT24C64,共 4 字节*****
```

*说明:将表号和用户电量共四字节数据写入 AT24C64 中

*入口参数:

* 1.数据间接寻址地址-buf

* 2.写入到 AT24C64 的地址字-addh,addrl

* 3.写入字节数-count

*出口参数:1 表示写成功,0 表示写失败

```
*****/
```

```
bit write_byte(unsigned char * buf,
               unsigned char addrh,
               unsigned char addrl,
               unsigned char count)
```

```
{
```

```
.....
```

```
}
```

d. 全局变量要注释其功能, 若为关键的局部变量同样需要注释其功能。

```
volatile UINT8 __ucSysMsg=SYS_IDLE;
```

```
void SYSSetMsgPriority(void)
```

```
{
```

```
SYSMMSG Msgt;//临时存储消息
```

```
UINT8 i;
```

```
}
```

e. 复杂的宏定义同样要加上注释。

```
/* SYS_MSG_MAP 建立一个消息映射
```

宏参数NAME: 消息映射表的名字

宏参数NUM_OF_MSG:消息映射的个数

```
*/
```

```
#define SYS_MSG_MAP(NAME, NUM_OF_MSG) do\
```

```
{\
```

```
    DEFINE_MSG_NAME((NAME));\
```

```
    UINT8 i;\
```

```
    for(i=0;i< NUM_OF_MSG;i++)\
```

```
    {\
```

```
        ININ_CUR_MSG(i)\
```

```
    }\
```

```
}while(0)
```

f. 复杂的结构体同样要加上注释。

```
/* 奇偶校验结构体*/
```

```
typedef struct _ PKT_PARITY
{
    UINT8 m_ucHead1; //首部1
    UINT8 m_ucHead2; //首部2
    UINT8 m_ucOptCode; //操作码
    UINT8 m_ucDataLength; //数据长度
    UINT8 m_szDataBuf[16]; //数据
    UINT8 m_ucParity; //奇偶校验值
}PKT_PARITY;
```

g. 相对独立的语句组注释。对这一组语句做特别说明，写在语句组上侧，和此语句组之间不留空行，与当前语句组的缩进一致。注意，说明语句组的注释一定要写在语句组上面，不能写在语句组下面。

3. 标识符

a. 变量的命名

方法一：采用匈牙利命名法。命名规则的主要思想是“在变量中加入前缀以增进人们对程序的理解”。

例如平时声明32位整型变量Length对应使用匈牙利命名法为unLength。现在列出经常用到的变量类型。

变量类型	示例
char	cLength
unsigned char	ucLength
short int	sLength
unsigned short int	usLength
int	nLength
unsigned int	unLength
char *	szBuf
unsigned char *	uszBuf
volatile unsigned char	__ucLength

方法二：

- 局部变量以小写字母命名；
- 全局变量以首字母大写方式命名（骆驼式）；
- 定义类型和宏定义常数以大写字母命名；
- 变量的作用域越大，它的名字所带有的信息就应该越多。
- 局部变量： `int student_age;`
- 全局变量： `int StudentAge;`
- 宏定义常数： `#define STUDENT_NUM 10`
- 类型定义： `typedef INT16S int;`

（我个人喜欢第二种方法）

b. 变量命名要注意缩写而且让人简单易懂，若是特别缩写要详细说明。

经常用到的缩写如：

Count 可缩写为Cnt

Message 可缩写为Msg

Packet 可缩写为Pkt

Temp 可缩写为Tmp

平时不经常用到的缩写，要注释：

SerialCommunication 可缩写为SrlComm //串口通信变量

SerialCommunicationStatus 可缩写为SrlCommStat //串口通信状态变量

c. 全局变量和全局函数的命名一定要详细，不惜多用几个单词，例如函数

UARTPrintfStringForLCD,

因为它们在整个项目的许多源文件中都会用到，必须让使用者明确这个变量或函数是干什么用的。局部变量和只在一个源文件中调用的内部函数的命名可以。简略一些，但不能太短，不要使用单个字母做变量名，只有一个例外：用*i*、*j*、*k* 做循环变量是可以的。

d. 用于编译开关的文件头，必须加上当前文件名称，防止编译时产生冲突。

例如在UARTInterface.h 头文件中，必须加上以下内容

```
#ifndef __UARTINTERFACE_H__
#define __UARTINTERFACE_H__
extern void UARTPrintfString(CONST INT8* str);
extern void UARTSendNBytes(UINT8 *ucSendBytes,UINT8 ucLen);
..... //其他外部声明的代码
#endif
```

e. 禁止用汉语拼音作为标识符名称，可读性极差。呵呵。

f. 建议名称间的区别要显而易见。使用标识符名称要注意的一个相关问题是发生在名称之间只有一个字符或少数字符不同的情况，特别是名称比较长时，当名称间的区别很容易被误读时问题就比较显著，比如1（数字1）和l（L 的小写）、0 和O、2 和Z、5 和S，或者n 和h。

4. 表达式和基本语句

a. 不要编写太复杂的复合表达式；

例如：

```
i = a >= b && c < d && c + f <= g + h; //复合表达式过于复杂
```

b. 不要有多用途的复合表达式；

例如：

```
d = (a = b + c) + r ; //应拆分为两个语句：
a = b + c;
d = a + r;
```

c. 如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。

例如：

```
if(a | b && a & c) //不良的风格
if((a | b) && (a & c)) //良好的风格
```

注意：只需记住加减运算的优先级低于乘除运算，其它地方一律加上括号。

d if 语句**d.a 布尔变量与零值比较**

不可将布尔变量直接与TRUE、FALSE 或者1、0 进行比较。

根据布尔类型的语义，零值为“假”（记为FALSE），任何非零值都是“真”（记为TRUE）。TRUE的值究竟是什么并没有统一的标准。例如Visual C++ 将TRUE 定义为1，而Visual Basic 则将TRUE 定义为-1。

例：假设布尔变量名字为flag，它与零值比较的标准if 语句如下：

```
if (flag)           // 表示flag为真时满足条件
if (!flag)          // 表示flag为假时满足条件
```

其它的用法都属于不良风格，例如：

```
if (flag == TRUE)
if (flag == 1 )
if (flag == FALSE)
if (flag == 0)
```

d.b 整型变量与零值比较

应当将整型变量用“==”或“!=”直接与0比较。

例：假设整型变量为value，它与零值比较的标准if 语句如下：

```
if (value == 0)
if (value != 0)
```

不可模仿布尔变量的风格而写成

```
if (value)           // 会让人误解 value 是布尔变量
if (!value)
```

小技巧：想必大家都有过将赋值操作符“=”当作比较相等操作符“==”用过，这个错误比较的隐晦，不易排查，而且编译器从不把这类事情当作是程序员犯下的错。避免的方法有两种，一种是养成良好的编程习惯，在比较数值时小心翼翼的处理；另一种方法见下面给出的代码：

```
if (NULL == p)
{
    .....
}
```

是不是觉得这种书写方式很古怪？不是程序写错了？

当然不是！

有经验的程序员为了防止将 **if (p == NULL)** 误写成 **if (p = NULL)**，而有意把p 和NULL 颠倒。编译器认为 **if (p = NULL)** 是合法的，但是会指出 **if (NULL = p)**是错误的，因为NULL不能被赋值。所以，再次遇到判断整型变量是否与某个数相等时，请这样写吧：

```
if (2==flag)
{
    .....
}
```

d.c 浮点变量与零值比较

不可将浮点变量用“==”或“!=”与任何数字比较。

千万要留意，无论float 还是double 类型变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该设法转化成“>”或“<”形式。

假设浮点变量的名字为x，应当将

```
if (x == 0.0) // 隐含错误的比较
```

转化为

```
if ((x>=-EPSINON) && (x<=EPSINON)) //EPSINON 是精度
```

5. 杂项

- a. 一些常量(如圆周率PI)或者常需要在调试时修改的参数最好用#define定义, 但要注意宏定义只是简单的替换, 因此有些括号不可少。
- b. 不要轻易调用某些库函数, 因为有些库函数代码很长(我是反对使用printf之类的库函数的, 但是是一家之言, 并不勉强各位)。
- c. 对各运算符的优先级有所了解, 记不得没关系, 加括号就是, 千万不要自作聪明说自己记得很牢。
- d. 不管有没有无效分支, switch函数一定要default这个分支。一来让阅读者知道程序员并没有遗忘default, 并且防止程序运行过程中出现的意外(健壮性)。
- e. 函数的参数和返回值没有的话最好使用void。
- f. 一些常数和表格之类的应该放到code中去以节省RAM。
- g. 程序编完编译看有多少code多少data, 注意不要使堆栈为难。
- h. 减少函数本身或函数间的递归调用
- i. 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。
- j. 在多重循环中, 应将最忙的循环放在最内层
- k. 避免循环体内含判断语句, 应将循环语句置于判断语句的代码块之中。
- l. 系统运行之初, 要初始化有关变量及运行环境, 防止未经初始化的变量被引用。
- m. 编写代码时要注意随时保存, 并定期备份, 防止由于断电、硬盘损坏等原因造成代码丢失。

二.模块化编程

当你在一个项目小组做一个相对较复杂的工程时,意味着你不再独自单干。你需要和你的小组成员分工合作,一起完成项目,这就要求小组成员各自负责一部分工程。比如你可能只是负责通讯或者显示这一块。这个时候,你就应该将自己的这一块程序写成一个模块,单独调试,留出接口供其它模块调用。最后,小组成员都将自己负责的模块写完并调试无误后,由项目组长进行组合调试。像这些场合就要求程序必须模块化。模块化的好处是很多的,不仅仅是便于分工,它还有助于程序的调试,有利于程序结构的划分,还能增加程序的可读性和可移植性。

初学者往往搞不懂如何模块化编程,其实它是简单易学,而且又是组织良好程序结构行之有效的方法之一。

本文将先大概讲一下模块化的方法和注意事项,最后将以初学者使用最广的keil c编译器为例,给出模块化编程的详细步骤。

模块化程序设计应该理解以下概述:

(1) 模块即是一个.c 文件和一个.h 文件的结合,头文件(.h)中是对于该模块接口的声明;

这一条概括了模块化的实现方法和实质:将一个功能模块的代码单独编写成一个.c文件,然后把该模块的接口函数放在.h文件中. 举例:假如你用到液晶显示,那么你可能会写一个液晶驱动模块,以实现字符、汉字和图像的现实,命名为:led_device.c, 该模块的.c文件大体可以写成:

```
/*
*                               液晶驱动模块
*
* 文    件: lcd_device.c
* 编 写 人: 小瓶盖
* 描    述: 液晶串行显示驱动模块, 提供字符、汉字、和图像的实现接口
* 编写时间: 2009. 07. 03
* 版    本: 1. 2
*/
#include ...
...
//定义变量
unsigned char flag;    //局部变量
static unsigned char value; //全局变量
...
//定义函数
//这是本模块第一个函数, 起到延时作用, 只供本模块的函数调用, 所以用到static关键字修饰
/******延时子程序******/
static void delay (uint us)    //delay time
{
}
//这是本模块的第二个函数, 要在其他模块中调用
/******写字符程序******/
** 功能: 向LCD写入字符
** 参数: dat_comm 为1写入的是数据, 为0写入的是指令
```

```

        content 为写入的数字或指令
*****
void wr_lcd (uchar dat_comm, uchar content)
{
.....
.....
/***** END Files*****/

```

注：此处只写出这两个函数，第一个延时函数的作用范围是模块内，第二个，它是其它模块需要的。为了简化，此处并没有写出函数体。

.h文件中给出模块的接口。在上面的例子中，向LCD写入字符函数：wr_lcd (uchar dat_comm, uchar content)就是一个接口函数，因为其它模块会调用它，那么.h文件中就必须将这个函数声明为外部函数（使用extern关键字修饰），另一个延时函数：void delay (uint us)只是在本模块中使用（本地函数，用static关键字修饰），因此它是不需要放到.h文件中的。

.h文件格式如下：

```

/*****
*
*          液晶驱动模块 头文件
*
* 文    件: lcd_device.h
* 编 写 人: 小瓶盖
* 编写时间: 2010.07.03
* 版    本: 1.0
*****
***/
//声明全局变量
extern unsigned char value;
//声明接口函数
extern void wr_lcd (uchar dat_comm, uchar content);    //向LCD写入字符
.....
/***** END Files*****/

```

这里注意三点：

1. 在keil 编译器中，extern这个关键字即使不声明，编译器也不会报错，且程序运行良好，但不保证使用其它编译器也如此。强烈建议加上，养成良好的编程规范。
2. .c文件中的函数只有其它模块使用时才会出现在.h文件中，像本地延时函数static void delay (uint us)即使出现在.h文件中也是在做无用功，因为其它模块根本不去调用它，实际上也调用不了它(static关键字的限制作用)。
3. 注意本句最后一定要加分号";"，相信有不少同学遇到过这个奇怪的编译器报错：**error C132: 'xxxx': not in formal parameter list**，这个错误其实是.h的函数声明的最后少了分号的缘故。

模块的应用：假如需要在LCD菜单模块lcd_menu.c中使用液晶驱动模块lcd_device.c中的函数void wr_lcd (uchar dat_comm, uchar content)，只需在LCD菜单模块的lcd_menu.c文件中加入液晶驱动模块的头文件lcd_device.h即可。

```

/*****
*
*          液晶菜单模块
*
* 文    件: lcd_menu.c
* 编 写 人: 小瓶盖
* 说    明: LCD菜单模块, 最多实现256级菜单, 与硬件无关。
* 编写时间: 2010.07.03
* 版    本: 1.0
*****/
#include "lcd_device.h" //包含液晶驱动程序头文件, 之后就可以在该.c文件中调用
                        //lcd_device.h中的全局函数, 使用液晶驱动程序里的全局
                        //变量(如果有的话)。

...
//调用向LCD写入字符函数
wr_lcd (0x01, 0x30);
...
//对全局变量赋值
value=0xff;
...

```

(2) 某模块提供给其它模块调用的外部函数及数据需在.h 文件中冠以extern 关键字声明;

这句话在上面的例子中已经有体现, 即某模块提供给其它模块调用的外部函数和全局变量需在.h 文件中冠以extern 关键字声明, 下面重点说一下全局变量的使用。使用模块化编程的一个难点(相对于新手)就是全局变量的设定, 初学者往往很难想通模块与模块公用的变量是如何实现的, 常规的做法就是本句提到的, 在.h文件中外部数据冠以extern关键字声明。比如上例的变量value就是一个全局变量, 若是某个模块也使用这个变量, 则和使用外部函数一样, 只需在使用的模块.c文件中包含#include "lcd_device.h"即可。

另一种处理模块间全局变量的方法来自于嵌入式操作系统uCOS-II, 这个操作系统处理全局变量的方法比较特殊, 也比较难以理解, 但学会之后妙用无穷, 这个方法只需用在头文件中定义一次。方法为:

在定义所有全局变量(uCOS-II将所有全局变量定义在一个.h文件内)的.h头文件中:

```

#ifndef xxx_GLOBALS
#define xxx_EXT
#else
#define xxx_EXT extern
#endif

```

.H 文件中每个全局变量都加上了xxx_EXT的前缀。xxx 代表模块的名字。

该模块的.C文件中有以下定义:

```

#define xxx_GLOBALS
#include "includes.h"

```

当编译器处理.C文件时, 它强制xxx_EXT(在相应.H文件中可以找到)为空, (因为xxx_GLOBALS已经定义)。所以编译器给每个全局变量分配内存空间, 而当编译器处理其他.C 文件时, xxx_GLOBAL没有定义, xxx_EXT 被定义为extern, 这样用户就可以调用外

部全局变量。为了说明这个概念，可以参见uC/OS_II.H,其中包括以下定义：

```
#ifndef OS_GLOBALS
#define OS_EXT
#else
#define OS_EXT extern
#endif
OS_EXT INT32U OSIdleCtr;
OS_EXT INT32U OSIdleCtrRun;
OS_EXT INT32U OSIdleCtrMax;
```

同时，uCOS_II.H 中有以下定义：

```
#define OS_GLOBALS
#include "includes.h"
```

当编译器处理uCOS_II.C 时，它使得头文件变成如下所示，因为OS_EXT 被设置为空。

```
INT32U OSIdleCtr;
INT32U OSIdleCtrRun;
INT32U OSIdleCtrMax;
```

这样编译器就会将这些全局变量分配在内存中。当编译器处理其他.C 文件时，头文件变成了如下的样子，因为OS_GLOBAL没有定义，所以OS_EXT 被定义为extern。

```
extern INT32U OSIdleCtr;
extern INT32U OSIdleCtrRun;
extern INT32U OSIdleCtrMax;
```

在这种情况下，不产生内存分配，而任何 .C文件都可以使用这些变量。这样的就只需在 .H文件中定义一次就可以了。

(3) 模块内的函数和全局变量需在.c 文件开头冠以static 关键字声明；

这句话主要讲述了关键字static的作用。Static是一个相当重要的关键字，他能对函数和变量做一些约束，而且可以传递一些信息。比如上例在LCD驱动模块.c文件中定义的延时函数static void delay (uint us)，这个函数冠以static修饰，一方面是限定了函数的作用范围只是在本模块中起作用，另一方面也给人传达这样的信息：该函数不会被其他模块调用。下面详细说一下这个关键字的作用，在C 语言中，关键字static 有三个明显的作用：

1. 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。
2. 在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。
3. 在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。

前两个都比较容易理解，最后一个作用就是刚刚举例中提到的延时函数（static void delay (uint us)），本地化函数是有相当好的作用的。

(4) 永远不要在.h 文件中定义变量！

呵呵，似乎有点危言耸听的感觉，但我想也不会有多少人会在.h文件中定义变量的。

比较一下代码：

代码一：

```
/*module1.h*/
int a = 5; /* 在模块1 的.h 文件中定义int a */
/*module1 .c*/
```

```
#include "module1.h" /* 在模块1 中包含模块1 的.h 文件 */
```

```
/*module2 .c*/
```

```
#include "module1.h" /* 在模块2 中包含模块1 的.h 文件 */
```

```
/*module3 .c*/
```

```
#include "module1.h" /* 在模块3 中包含模块1 的.h 文件 */
```

以上程序的结果是在模块1、2、3 中都定义了整型变量a，a 在不同的模块中对应不同的地址元，这个世界上从来不需要这样的程序。正确的做法是：

代码二：

```
/*module1.h*/
```

```
extern int a; /* 在模块1 的.h 文件中声明int a */
```

```
/*module1 .c*/
```

```
#include "module1.h" /* 在模块1 中包含模块1 的.h 文件 */
```

```
int a = 5; /* 在模块1 的.c 文件中定义int a */
```

```
/*module2 .c*/
```

```
#include "module1.h" /* 在模块2 中包含模块1 的.h 文件 */
```

```
/*module3 .c*/
```

```
#include "module1.h" /* 在模块3 中包含模块1 的.h 文件 */
```

这样如果模块1、2、3 操作a 的话，对应的是同一片内存单元。

注：

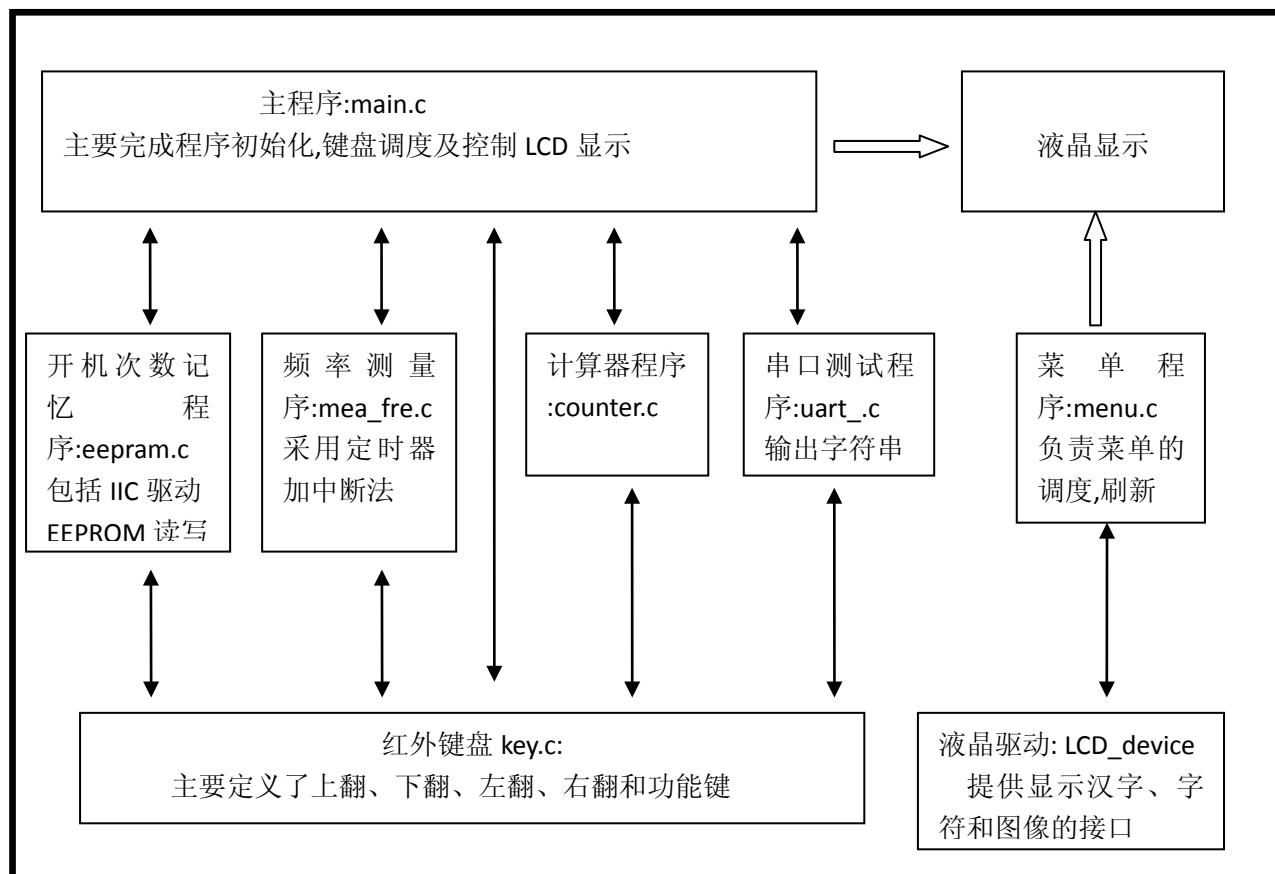
一个嵌入式系统通常包括两类（注意是两类，不是两个）模块：

- （1）硬件驱动模块，一种特定硬件对应一个模块；
- （2）软件功能模块，其模块的划分应满足低耦合、高内聚的要求。

下面以 keil C 编译器为例，讲一下模块化编程的步骤。

下面这个程序分为三层，共 7 个模块，共同为主程序服务（它们之间也会相互调用）。

程序的结构图如下所示：



程序主要模块和功能简介：

一. 底层驱动

1. 红外键盘:程序通过红外键盘进行操作。红外键盘独占定时器 0 和外部中断 0，以实现红外解码和键盘键值的识别。红外键盘定义了五个按键，分别为上翻、下翻、左翻、右翻和确认键。
2. LCD 液晶显示: 程序主要通过 LCD 显示信息，LCD 液晶显示驱动提供显示汉字、图形和 ASCII 码的函数接口。可以全屏、单行显示汉字，任意位置显示 ASCII 码，还可以全屏、半屏显示图形。

二. 功能模块

1. LCD 菜单程序: 菜单程序可以使人机交互更加方便、容易。本菜单程序的菜单级别深度受 RAM 大小的限制，每增加一级菜单将多消耗 4 字节的 RAM。菜单程序主要完成菜单功能函数的调度，LCD 显示刷新。
2. 计算器程序: 实现 65536 以内的加、减、乘、除，超出范围会出现溢出，溢出发生时，LCD 显示“错误：出现溢出”的错误提示，同时本次运算被忽略。对于负数

会显示“-”号，除数为零时 LCD 显示“错误：除数为零”的错误提示。

3. 开机次数记忆程序:主要对基于 IIC 总线的 EEPROM 进行读写,单片机每次上电后,将开机次数写入 EEPROM.

4. 串口测试程序:进入该程序后,单片机向电脑发送字符串“Hello Word!”,发送数字 24 (以字符的形式显示)。编写此程序的目的是为了能够方便的向电脑发送字符串和变量,便于程序的调试。串口占用串口资源,与频率测量程序共享定时器 1

5. 频率测量:复用定时器 1, 占用外部中断 1, 实现 5~20KHZ 频率的测量。

三. 主程序

主程序主要完成程序的初始化, LCD 菜单显示, 监视键盘程序并根据键值更新菜单。

步骤为:

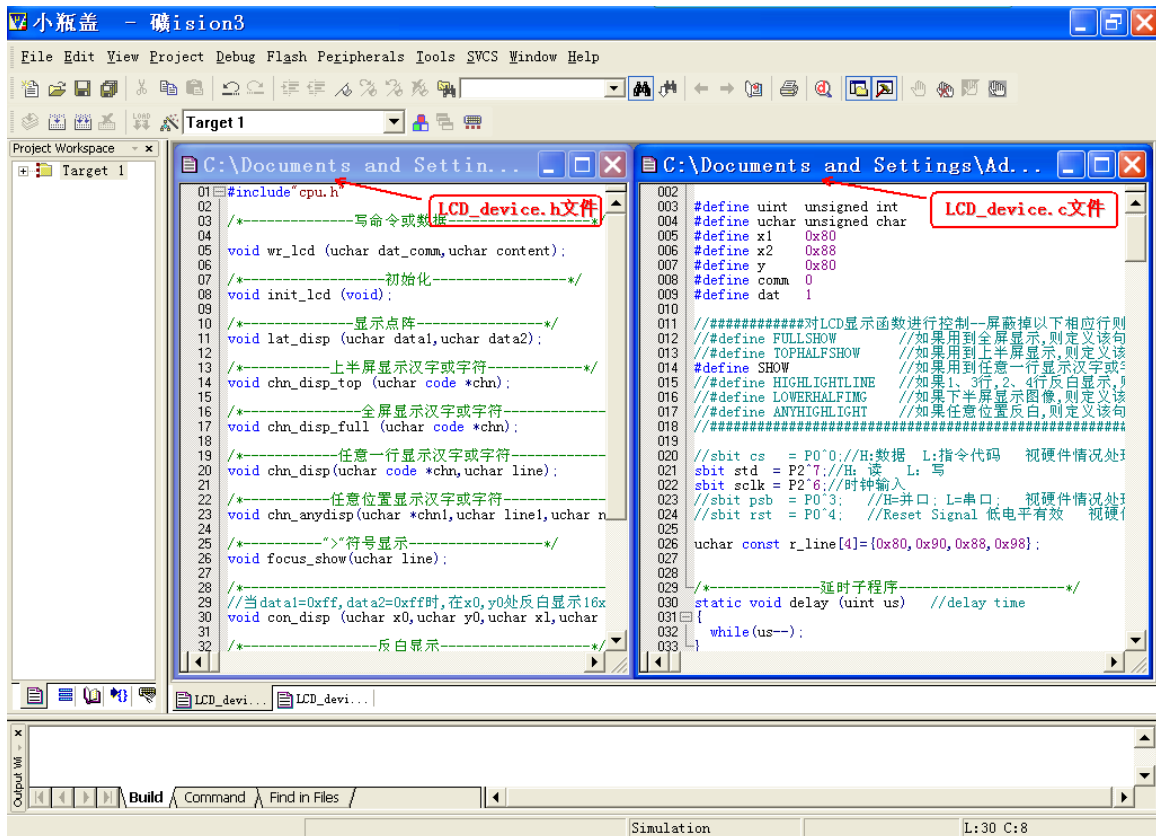
1.新建工程。

2.点击 File—New (或者点击快捷图标: ) , 新建一个文档。

3.点击 File—Save (或者点击快捷图标: ) , 保存新建的文档, 在文件名后填写 LCD_device.c (液晶驱动模块: LCD_device, 提供显示汉字、字符和图像的接口), 点击确定。在该文档内编写 LCD 驱动程序。

4. 点击 File—New (或者点击快捷图标: ) , 再新建一个文档。

5. 点击 File—Save (或者点击快捷图标: ) , 保存新建的文档, 在文件名后填写 LCD_device.h (液晶驱动模块的头文件, 模块的接口和全局变量在这里定义)。点击确定。在该文档中整理全局变量和接口函数。以上步骤之后的效果见下图:



至此，液晶驱动模块书写完毕，可以对这个模块单独的调试。

6.重复以上步骤 2~5，定义 红外键盘模块: key.c 与 key.h

菜单模块: menu.c 与 menu.h

串口通信模块: uart_.c 与 uart.h

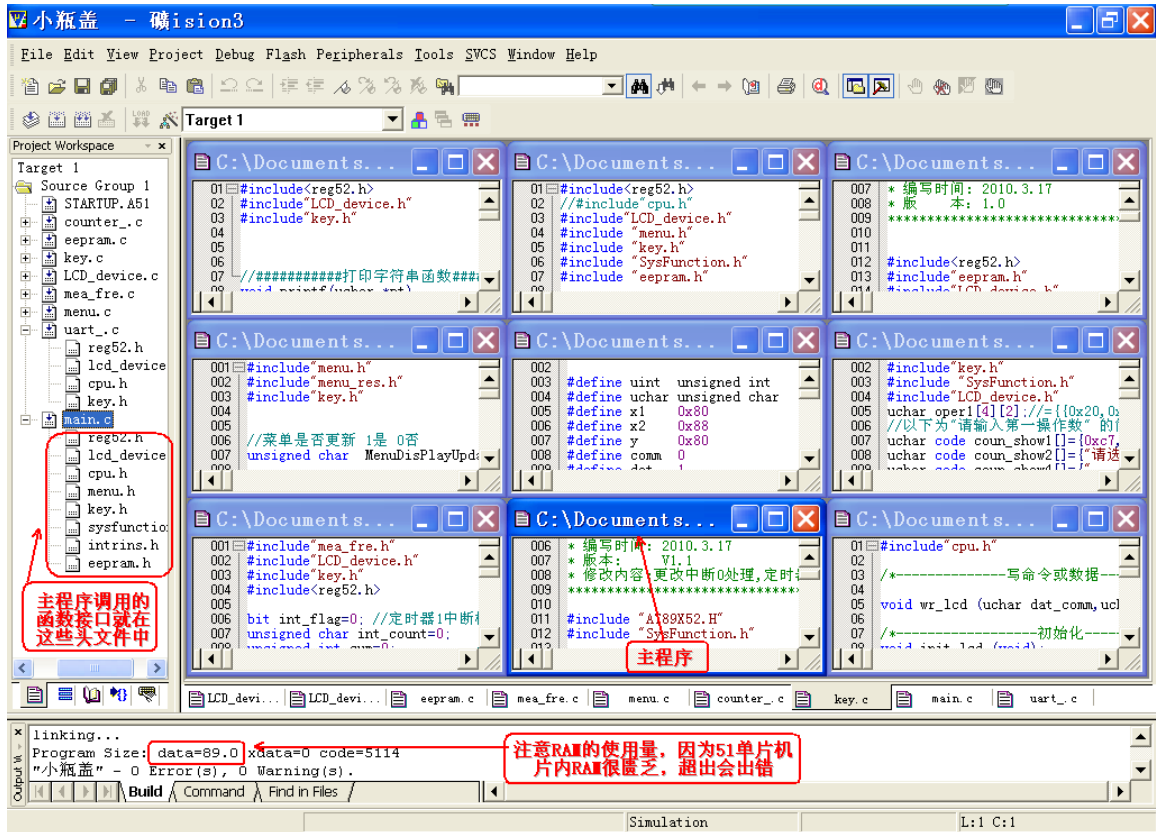
计算器模块: counter.c 与 counter.h

频率测量模块: mea_fre.c 与 mea_fre.h

开机次数记忆模块: eepram.c 与 eepram.h

7.重复以上步骤 2~3，定义主程序 main.c

最终效果如下图所示:



完成 1~7 个步骤后，有些小白就习惯性的点击编译按钮了，这时候会出现两个警告信息：

*** WARNING L1: UNRESOLVED EXTERNAL SYMBOL

*** WARNING L2: REFERENCE MADE TO UNRESOLVED EXTERNAL

这是因为你只是编写好了程序模块，却没有把他们加入到工程的缘故。

解决方法：在 Project Workspace 框中，右击 Source group 1 文件夹，选择 Add Files to Group 'Source Group 1'，在弹出的对话框中添加你的.c 文件即可。

遥想很久很久以前，我也对上面的两个警告有过亲身体会。那时候我还在大学，周围有一大群的好哥们一起玩，想逃课逃课，想睡觉睡觉，现在...哎呀，泪流满面啊!!!

三.不可不用的关键字

1.static关键字

这个关键字前面也有提到，它的作用是强大的。

要对static关键字深入了解，首先需要掌握标准C程序的组成。

标准C程序一直由下列部分组成：

1) 正文段——CPU执行的机器指令部分，也就是你的程序。一个程序只有一个副本；只读，这是为了防止程序由于意外事故而修改自身指令；

2) 初始化数据段（数据段）——在程序中所有赋了初值的全局变量，存放在这里。

3) 非初始化数据段（bss段）——在程序中没有初始化的全局变量；内核将此段初始化为0。

注意：只有全局变量被分配到数据段中。

4) 栈——增长方向：自顶向下增长；自动变量以及每次函数调用时所需要保存的信息（返回地址；环境信息）。这句很关键，常常有笔试题会问到什么东西放到栈里面就足以说明。

5) 堆——动态存储分配。

在嵌入式C语言当中，它有三个作用：

作用一：在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。

这样定义的变量称为局部静态变量：在局部变量之前加上关键字static，局部变量就被定义成为一个局部静态变量。也就是上面的作用一中提到的在函数体内定义的变量。除了类型符外，若不加其它关键字修饰，默认都是局部变量。比如以下代码：

```
void test1 (void)
{
    unsigned char a;
    static unsigned char b;
    ...
    a++;
    b++;
}
```

在这个例子中，变量a是局部变量，变量b为局部静态变量。作用一说明了局部静态变量b的特性：在函数体，一个被声明为静态的变量（也就是局部静态变量）在这一函数被调用过程中维持其值不变。这句话什么意思呢？若是连续两次调用上面的函数test1：

```
void main (void)
{
    ...
    test1 ();
    test1 ();
    ...
}
```

然后使程序暂停下来，读取a和b的值，你会发现，a=1，b=2。怎么回事呢，每次调用test1函数，局部变量a都会重新初始化为0x00；然后执行a++；而局部静态变量在调用过程中却能维持其值不变。

通常利用这个特性可以统计一个函数被调用的次数。

声明函数的一个局部变量，并设为static类型，作为一个计数器，这样函数每次被调用的时候就可以进行计数。这是统计函数被调用次数的最好的办法，因为这个变量是和函数息息相关的，而函数可能在多个不同的地方被调用，所以从调用者的角度来统计比较困难。代码如下：

```
void count();
int main()
{
    int i;
    for (i = 1; i <= 3; i++)
    {
        count();
    }
    return 0;
}
void count()
{
    static num = 0;
    num++;
    printf(" I have been called %d", num, "times\n");
}
```

输出结果为：

```
I have been called 1 times.
I have been called 2 times.
I have been called 3 times.
```

看一下局部静态变量的详细特性，注意它的作用域。

1) 内存中的位置：静态存储区

2) 初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3) 作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域随之结束。

注：当static用来修饰局部变量的时候，它就改变了局部变量的存储位置，从原来的栈中存放改为静态存储区。但是局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问。

作用二：在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。

这样定义的变量也称为全局静态变量：在全局变量之前加上关键字static，全局变量就被定义成为一个全局静态变量。也就是上述作用二中提到的在模块内（但在函数体外）声明的静态变量。

定义全局静态变量的好处：

<1>不会被其他文件所访问，修改，是一个本地的局部变量。

<2>其他文件中可以使用相同名字的变量，不会发生冲突。

全局变量的详细特性，注意作用域，可以和局部静态变量相比较：

1) 内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

2) 初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3) 作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。

当static用来修饰全局变量的时候，它就改变了全局变量的作用域（在声明他的文件之外是不可见的），但是没有改变它的存放位置，还是在静态存储区中。

作用三：在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。

这样定义的函数也成为静态函数：在函数的返回类型前加上关键字static，函数就被定义成为静态函数。函数的定义和声明默认情况下是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

定义静态函数的好处：

<1> 其他文件中可以定义相同名字的函数，不会发生冲突

<2> 静态函数不能被其他文件所用。它定义一个本地的函数。

这里我一直强调数据和函数的本地化，这对于程序的结构甚至优化都有巨大的好处，更大的作用是，本地化的数据和函数能给人传递很多有用的信息，能约束数据和函数的作用范围。在C++的对象和类中非常注重的私有和公共数据/函数其实就是本地和全局数据/函数的扩展，这也从侧面反应了本地化数据/函数的优势。

最后说一下存储说明符，在标准C语言中，存储说明符有以下几类：

auto、register、extern和static

对应两种存储期：自动存储期和静态存储期。

auto和register对应自动存储期。具有自动存储期的变量在进入声明该变量的程序块时被建立，它在该程序块活动时存在，退出该程序块时撤销。

关键字extern和static用来说明具有静态存储期的变量和函数。用static声明的局部变量具有静态存储持续期（static storage duration），或静态范围（static extent）。虽然他的值在函数调用之间保持有效，但是其名字的可视性仍限制在其局部域内。**静态局部对象在程序执行到该对象的声明处时被首次初始化。**

2. const 关键字

const关键字也是一个优秀程序中经常用到的关键字。关键字const 的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。通过给优化器一些附加的信息，使用关键字const 也许能产生更紧凑的代码。合理地使用关键字const 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少bug的出现。

深入理解const关键字，你必须知道：

a. const关键字修饰的变量可以认为有只读属性，但它绝不与常量划等号。如下代码：

```
const int i=5;
int j=0;
```

...

i=j; //非法，导致编译错误，因为只能被读**j=i; //合法**

b. **const**关键字修饰的变量在声明时必须进行初始化。如下代码：

const int i=5; //合法**const int j; //非法，导致编译错误**

c. 用**const**声明的变量虽然增加了分配空间，但是可以保证类型安全。**const**最初是从C++变化得来的，它可以替代**define**来定义常量。在旧版本(标准前)的c中，如果想建立一个常量，必须使用预处理器：

#define PI 3.14159

此后无论在何处使用**PI**，都会被预处理器以**3.14159**替代。编译器不对**PI**进行类型检查，也就是说可以不受限制的建立宏并用它来替代值，如果使用不慎，很可能由预处理引入错误，这些错误往往很难发现。而且，我们也不能得到**PI**的地址（即不能向**PI**传递指针和引用）。**const**的出现，比较好的解决了上述问题。

d. C标准中，**const**定义的常量是全局的。

e. 必须明白下面语句的含义，我自己是反复记忆了许久才记住，方法是：若是想定义一个只读属性的指针，那么关键字**const**要放到‘*’后面。

char *const cp; //指针不可改变，但指向的内容可以改变**char const *pc1; //指针可以改变，但指向的内容不能改变****const char *pc2; //同上（后两个声明是等同的）**

f. 将函数传入参数声明为**const**，以指明使用这种参数仅仅是为了效率的原因，而不是想让调用函数能够修改对象的值。

参数**const**通常用于参数为指针或引用的情况，且只能修饰输入参数；若输入参数采用“值传递”方式，由于函数将自动产生临时变量用于复制该参数，该参数本就不需要保护，所以不用**const**修饰。例子：

void fun0(const int * a);**void fun1(const int & a);**

调用函数的时候，用相应的变量初始化**const**常量，则在函数体中，按照**const**所修饰的部分进行常量化，如形参为**const int * a**，则不能对传递进来的指针所指向的内容进行改变，保护了原指针所指向的内容；如形参为**const int & a**，则不能对传递进来的引用对象进行改变，保护了原对象的属性。

g. 修饰函数返回值，可以阻止用户修改返回值。（在嵌入式C中一般不用，主要用于C++）

h. **const**消除了预处理器的值替代的不良影响，并且提供了良好的类型检查形式和安全性，在可能的地方尽可能的使用**const**对我们的编程有很大的帮助，前提是：你对**const**有了足够的理解。

最后，举两个常用的标准C库函数声明，它们都是使用**const**的典范。

1.字符串拷贝函数：**char *strcpy (char *strDest, const char *strSrc) ;**

2.返回字符串长度函数：**int strlen (const char *str) ;**

3. volatile关键字

一个定义为**volatile** 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这

个变量的值，而不是使用保存在寄存器里的备份。

由于访问寄存器的速度要快过RAM，所以编译器一般都会作减少存取外部RAM的优化。比如：

```
static int i=0;

int main(void)
{
    ...
    while (1)
    {
        if (i)
            dosomething();
    }
}

/* Interrupt service routine. */
void ISR_2(void)
{
    i=1;
}
```

程序的本意是希望ISR_2中断产生时，在main当中调用dosomething函数，但是，由于编译器判断在main函数里面没有修改过i，因此可能只执行一次对从i到某寄存器的读操作，然后每次if判断都只使用这个寄存器里面的“i副本”，导致dosomething永远也不会被调用。

如果将变量加上volatile修饰，则编译器保证对此变量的读写操作都不会被优化（肯定执行）。此例中i也应该如此说明。

一般说来，volatile用在如下的几个地方：

- 1、中断服务程序中修改的供其它程序检测的变量需要加volatile；
 - 2、多任务环境下各任务间共享的标志应该加volatile；
 - 3、存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能有不同的意义；
- 不懂得volatile 的内容将会带来灾难,这也是区分C语言和嵌入式C语言程序员的一个关键因素。为强调volatile的重要性,再次举例分析:

代码一：

```
int a,b,c;
//读取I/O空间0x100端口的内容
a= inword(0x100);
b=a;
a=inword(0x100)
```

```
c=a;
```

代码二:

```
volatile int a;
int a,b,c;
//读取I/O空间0x100端口的内容
a= inword(0x100);
b=a;
a=inword(0x100)
c=a;
```

在上述例子中,代码一会被绝大多数编译器优化为如下代码:

```
a=inword(0x100)
b=a;
c=a;
```

这显然与编写者的目的不相符,会出现I/O空间0x100端口漏读现象,若是增加volatile,像代码二所示的那样,优化器将不会优化掉任何代码.

从上面来看,volatile关键字是会降低编译器优化力度的,但它保证了程序的正确性,所以在适合的地方使用关键字volatile是件考验编程功底的事情.

4.struct与typedef关键字

面对一个人的大型C/C++程序时,只看其对struct的使用情况我们就可以对其编写者的编程经验进行评估。因为一个大型的C/C++程序,势必要涉及一些(甚至大量)进行数据组合的结构体,这些结构体可以将原本意义属于一个整体的数据组合在一起。从某种程度上来说,会不会用struct,怎样用struct是区别一个开发人员是否具备丰富开发经验的标志。

在网络协议、通信控制、嵌入式系统的C/C++编程中,我们经常要传送的不是简单的字节流(char型数组),而是多种数据组合起来的一个整体,其表现形式是一个结构体。

经验不足的开发人员往往将所有需要传送的内容依顺序保存在char型数组中,通过指针偏移的方法传送网络报文等信息。这样做编程复杂,易出错,而且一旦控制方式及通信协议有所变化,程序就要进行非常细致的修改。

用法:

在C中定义一个结构体类型要用typedef:

```
typedef struct Student
{
    int a;
}Stu;
```

于是在声明变量的时候就可: Stu stu1;

如果没有typedef就必须用struct Student stu1;来声明

这里的Stu实际上就是struct Student的别名。

另外这里也可以不写Student(于是也不能struct Student stu1;了)

```
typedef struct
{
    int a;
}Stu;
```

struct关键字的一个总要作用是它可以实现对数据的封装,有一点点类似与C++的对象,

可以将一些分散的特性对象化,这在编写某些复杂程序时提供很大的方便性.

比如编写一个菜单程序,你要知道本级菜单的菜单索引号、焦点在屏上是第几项、显示第一项对应的菜单条目索引、菜单文本内容、子菜单索引、当前菜单执行的功能操作。若是对上述条目单独操作,那么程序的复杂程度将会大到不可想象,若是菜单层数少些还容易实现,一旦菜单层数超出四层,呃~我就没法形容了。若是有编写过菜单程序的朋友或许理解很深。这时候结构体struct就开始显现它的威力了:

//结构体定义

typedef struct

{

unsigned char CurrentPanel;**//本级菜单的菜单索引号**

unsigned char ItemStartDisplay;**//显示第一项对应的菜单条目索引**

unsigned char FocusLine;**//焦点在屏上是第几项**

}Menu_Statestruct;

typedef struct

{

unsigned char *MenuTxt;**//菜单文本内容**

unsigned char MenuChildID;**//子菜单索引**

void (*CurrentOperate)();**//当前菜单执行的功能操作**

}MenuItemStruct;

typedef struct

{

MenuItemStruct *MenuPanelItem;

unsigned char MenuItemCount;

}MenuPanelStruct;

这里引用我巩师兄所写的菜单程序中的结构体定义,这个菜单程序最大可以到256级菜单。我当初要写一个菜单程序之前,并没有对结构体了解多少,也没有想到使用结构体。只是一层层菜单的单独处理:如果按键按下,判断是哪个按键,调用对应按键的子程序,刷屏显示。这样处理起来每一层都要判断当前的光标所在行,计算是不是在显示屏的顶层,是不是在显示层的底层,是不是需要翻页等等,非常的繁琐。后来在网上找资料,就找到了我师兄编写的这个程序,开始并不知道是他写的,在看源程序的时候看到了作者署名:中国传惠TranSmart,才知道是他。花了一天的时间阅读代码和移植,才知道结构体原来有这么个妙用,当定义了上述三个结构体之后,菜单程序结构立刻变的简单很多,思路也无比的清晰起来。

四.MISRA: 2004 - 安全编程

程序经常长期运行，其运行结果与预期不符，或者调试之初就发现根本无法运行，这种现象可以说是程序出现错误，错误的出现反应了程序的风险性，风险出现的原因有很多，MISRA:2004认为C程序设计中存在的风险可能由5个方面造成：程序员的失误、程序员对语言的误解、程序员对编译器的误解、编译器的错误和运行出错(runtime errors)。

可以看出，程序员本身的综合能力是程序风险出现或降低的主导。

程序员的失误是极其常见且难以避免的，程序员也是人。有些错误编译器可以很好的将其找出，并提示程序员进行修改，这类错误可归结为语法错误。毫无疑问，这类错误是最容易找出并修改的。若是你还在为能正确的编译程序而烦恼，那么，关于基本的C语言语法以及编译器调试信息是你当前应该首先掌握的。然而，有更多的错误，编译器是无法给出提示的，比如">"与">="的差别，这类错误可以成为逻辑错误。这类错误的大部分是最初很难发现的。

可能任何入门的程序员都有过把">"当成">="的经历，像：

```
If(flag=0x01) ;(而你的本意是想判断flag是否等于0x01)
```

这种错误编译器从来不认为是程序员的失误。

C语言是灵活多变的，是非常自由的，这种特性也使得真正对C语言的规则彻底理解困难了很多。程序员对C语言或者是编译器的理解失误造成的错误也非常的普遍。比如：

```
if ( ishigh && (x == flag++))
```

很多程序员认为执行了这条指令后，flag变量的值就会自动加1。但真实的情况如何呢？

若是ishigh为逻辑0，&&后面的表达式就不会被运行，因为整个表达式的结果一定是逻辑0。有一条规则：逻辑运算符&&或||的右操作数不得带有副作用（side effect）*，就是为了避免这种情况下可能出现的问题。

*所谓带有副作用，就是指执行某条语句时会改变运行环境，如执行x=flag++之后，flag的值会发生变化。

还有些错误是由编译器（或者说是编写编译器的程序员）本身造成的。这些错误往往较难发现，有可能会一直存留在最后的程序中。

运行错误指的是那些在运行时出现的错误，如除数等于零、指针地址无效等问题。运行错误在语法检查时一般无法发现，但一旦发生很可能导致系统崩溃。

为了使嵌入式程序更安全，汽车工业软件可靠性联合会（以下简称MISRA）于1998年发布了一个针对汽车工业软件安全性的C语言编程规范——《汽车专用软件的C语言编程指南》，共有127条规则，称为MISRA:1998。2004年，MISRA出版了该规范的新版本——MISRA:2004。在新版本中，还将面向的对象由汽车工业扩大到所有的高安全性要求

（Critical）系统。在MISRA:2004中，共有强制规则121条，推荐规则20条，并删除了15条旧规则。任何符合MISRA:2004编程规范的代码都应该严格的遵循121条强制规则的要求，并应该在条件允许的情况下尽可能符合20条推荐规则。

本文对其中所有的规则给予说明，并在括号内附英文原文，以作参考。

一.开发环境 (Environment)

规则 1.1（强制）： 所有代码都必须遵照ISO 9899:1990 “Programming languages - C”，由ISO/IEC 9899/COR1:1995，ISO/IEC 9899/AMD1:1995，和ISO/IEC9899/COR2:1996 修订。

(1.1 All code shall conform to ISO 9899:1990“Programming languages – C”, amended and c

corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.)

之所以没有用更新的ISO1999规则,是因为这些规则过于庞大,并且去掉了很多底层硬件的操作,所以面向嵌入式C语言,最多的是使用ISO1990规则,这与嵌入式离不开硬件是相符合的。

规则 1.2 (强制): 不能有对未定义行为或未指定行为的依赖性。

(1.2 No reliance shall be placed on undefined or unspecified behaviour.)

这项规则要求任何对未定义行为或未指定行为的依赖,除非在其他规则中做了特殊说明,都应该避免。如果其他某项规则中声明了某个特殊行为,那么就只有这项特定规则在其需要时给出背离性。

规则 1.3 (强制): 多个编译器和/或语言只能在为语言/编译器/汇编器所适合的目标代码定义了通用接口标准时使用。

(1.3 Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.)

如果一个模块是以非C语言实现的或是以不同的C编译器编译的,那么必须要保证该模块能够正确地同其他模块集成。C语言行为的某些特征依赖于编译器,于是这些行为必须能够为使用的编译器所理解。例如:栈的使用、参数的传递和数据值的存储方式(长度、排列、别名、覆盖,等等)。

规则 1.4 (强制): 编译器/链接器要确保31个有效字符和大小写敏感能被外部标识符支持。

(1.4 The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.)

必须检查你的编译器/链接器是否具有这种特性,如果编译器/链接器不能满足这种限制,就使用编译器本身的约束。即,你要明白你使用的编译器/连接器最大能支持定义变量的有效字符个数。举例: `int abcdefg;`

`int abcdefgh;`

若你的编译器/连接器只支持所定义变量的有效字符个数为6,那么,以上两个变量的定义将是重复的,因为,编译器/连接器只识别到 `int abcdef;` 多余的字符将被忽略,这是件相当可怕的事情,且是很难排查的错误,这时候,你所定义的变量字符个数就不能多余6个,也就是使用编译器本身的约束。好在现在绝大多数(至少我没有见过支持的有效字符少于31个C编译器/连接器)的C编译器都是至少支持31个有效字符的。

规则 1.5 (建议): 浮点应用应该适应于已定义的浮点标准

(1.5 Floating-point implementations should comply with a defined floating-point standard.)

浮点运算会带来许多问题,一些问题(而不是全部)可以通过适应已定义的标准来克服。其中一个合适的标准是 ANSI/IEEE Std 754

二.语言外延 (Language Extensions)

规则 2.1（强制）： 汇编语言应该被封装并隔离。

(2.1 Assembly language shall be encapsulated and isolated.)

在需要使用汇编指令的地方，建议以如下方式封装并隔离这些指令：

(a) 汇编函数、(b) C函数、(c) 宏。

出于效率的考虑，有时必须要嵌入一些简单的汇编指令，如开关中断。如果不管出于什么原因需要这样做，那么最好使用宏来完成。

```
#define NOP asm ("NOP");
```

规则2.2（强制）： 源代码应该使用 /*...*/ 类型的注释。

(2.2 Source code shall only use /* ... */ style comments.)

这排除了如 // 这样C99 类型的注释和C++类型的注释，因为它在C90 中是不允许的。许多编译器支持 // 类型的注释以做为对C90 的扩展。预处理指令（如#define）中 // 的使用可以改变，/*...*/和//的混合使用也是不一致的。这不仅是类型问题，因为不同的编译器（在C99之前）可能会有不同的行为。

规则 2.3（强制）： 字符序列 /* 不应出现在注释中。

(2.3 The character sequence /* shall not be used within a comment.)

C 不支持注释的嵌套，尽管一些编译器支持它以做为语言扩展，但谁又能保证所有编译器都支持注释嵌套且你的程序不会移植到这些编译器上编译呢。一段注释以/*开头，直到第一个*/为止，在这当中出现的任何/*都违反了本规则。考虑如下代码段：

```
/* some comment, end comment marker accidentally omitted
```

```
<<New Page>>
```

```
Perform_Critical_Safety_Function (X);
```

```
/* this comment is not compliant */
```

在检查包含函数调用的页中，假设它是可执行代码。

因为可能会省略掉注释的结束标记，那么对安全关键函数的调用将不会被执行。

规则 2.4（建议）： 代码段不应被“注释掉”（comment out）

(2.4 Sections of code should not be “commented out”.)

当源代码段不需要被编译时，应该使用条件编译来完成（如带有注释的#if 或#ifdef 结构）。为这种目的使用注释的开始和结束标记是危险的，因为C 不支持嵌套的注释，而且已经存在于代码段中的任何注释将影响执行的结果。

三. 文档化(Documentation)

规则 3.1（强制）： 所有现定义的（implementation-defined）行为的使用都应该文档化。

(3.1 All usage of implementation-defined behaviour shall be documented.)

本条例强调了书写文档的重要性，特别是自己定义的一些行为更要以文档的形式记录下来。比如某日对代码的添加，修改，删除等等，都要用文档加以说明。

规则 3.2（强制）： 字符集和相应的编码应该文档化。

(3.2 The character set and the corresponding encoding shall be documented.)

例如，ISO 10646定义了字符集映射到数字值的国际标准。

出于可移植性的考虑，字符常量和字符串只能包含映射到已经文档化的子集中的字符。

规则3.3（建议）： 应该确定、文档化和重视所选编译器中整数除法的实现。

(3.3 The implementation of integer division in the chosen compiler should be determined, documented and taken into account.)

当两个有符号整型数做除法时，ISO 兼容的编译器的运算可能会为正或为负。首先，它可能以负余数向上四舍五入（如， $-5/3 = -1$ ，余数为 -2 ），或者可能以正余数向下四舍五入（如， $-5/3 = -2$ ，余数为 $+1$ ）。重要的是要确定这两种运算中编译器实现的是哪一种，并以文档方式提供给编程人员，特别是第二种情况（通常这种情况比较少）。

规则 3.4（强制）： 所有`#pragma` 指令的使用应该文档化并给出良好解释。

(3.4 All uses of the `#pragma` directive shall be documented and explained.)

这项规则为本文档的使用者提供了产生其应用中使用的任何`pragma` 的要求。每个`pragma`的含义要写成文档，文档中应当包含完全可理解的对`pragma` 行为及其在应用中之含义的充分描述。

应当尽量减少任何 `pragma` 的使用，尽可能地把它本地化和封装成专门的函数。

规则 3.5（强制）： 如果做为其他特性的支撑，实现定义（**implementation-defined**）的行为和位域（**bitfields**）集合应当文档化。

(3.5 If it is being relied upon, the implementation defined behaviour and packing of bitfields shall be documented.)

这是在使用了规则6.4 和规则6.5 中描述的非良好定义的位域时遇到的特定问题。C 当中的位域是该语言中最缺乏良好定义的部分之一。位域的使用可能体现在两个主要方面：

□ 为了在大的数据类型（同`union` 一起）中访问独立的数据位或成组数据位。该用法是不允许的（见规则 18.4）。

□ 为了访问用于节省存储空间而打包的标志（**flags**）或其他短型（**short-length**）数据。为了有效利用存储空间而做的短型数据的打包，是本文档所设想的唯一可接受的位域使用。假定结构元素只能以其名字来访问，那么程序员就无需设想结构体中位域的存储方式。

我们建议结构的声明要保持位域的设置，并且在同一个结构中不得包含其他任何数据。要注意的是，在定义位域的时候不需要追随规则6.3，因为它们的长度已经定义在结构中了。如果编译器带有一个开关以强制位域遵循某个特定的布局，那么它有助于下面的判断。

例如下面可接受的代码：

```
struct message /* Struct is for bit-fields only */
{
    signed int little: 4; /* Note: use of basic types is required */
    unsigned int x_set: 1;
    unsigned int y_set: 1;
}message_chunk;
```

如果要使用位域，就得注意实现定义的行为所存在的领域及其潜藏的缺陷（意即不可移植性）。特别地，程序员应当注意如下问题：

□ 位域在存储单元中的分配是实现定义（**implementation-defined**）的，也就是说，它们在存储单元（通常是一个字节）中是高端在后（**high end**）还是低端在后（**low end**）的。

□ 位域是否重叠了存储单元的界限同样是实现定义的行为（例如，如果顺序存储一个6位的域和一个4位的域，那么4位的域是全部从新的字节开始，还是其中2位占据一个字节

中的剩余2 位而其他2 位开始于下个字节)。

规则 3.6 (强制)： 产品代码中使用的所有库都要适应本文档给出的要求，并且要经过适当的验证。

(3.6 All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.)

本规则的对象是产品代码中的任意库，因此这些库可能包含编译器提供的标准库、其他第三方的库或者实验室中自己开发的库。

4 字符集

规则 4.1 (强制)： 只能使用 ISO 标准定义的字符集。

(4.1 Only those escape sequences that are defined in the ISO C standard shall be used.)

规则 4.2 (强制)： 不能使用三元组序列 (trigraphs)。

(Trigraphs shall not be used)

三元组序列由 2 个问号序列后跟1 个确定字符组成 (如, ??- 代表 “~” (非) 符号, 而??) 代表 “]” 符号)。它们可能会对2 个问号标记的其他使用造成意外的混淆, 例如字符串“(Date should be in the form ??-??-??)”将不会表现为预期的那样, 实际上它被编译器解释为“(Date should be in the form ~~)”

5 标识符

规则 5.1 (强制)： 标识符 (内部的和外部的) 的有效字符数不能多于 31。

(5.1 Identifiers (internal and external) shall not rely on the significance of more than 31 characters.)

ISO 标准要求在内部标识符之间前31 个字符必须是不同的以保证可移植性。即使编译器支持, 也不能超出这个限制。

ISO 标准要求外部标识符之间前6 个字符必须是不同的 (忽略大小写) 以保证最佳的可移植性。然而这条限制相当严格并被认为是必须的。本规则的意图是为了在一定程度上放宽ISO 标准的要求以适应当今的环境, 但应当确保31 个字符/大小写的有效性是可以由实现所支持的。

使用标识符名称要注意的一个相关问题是发生在名称之间只有一个字符或少数字符不同的情况, 特别是名称比较长时, 当名称间的区别很容易被误读时问题就比较显著, 比如1 (数字1) 和l (L 的小写)、0 和O、2 和Z、5 和S, 或者n 和h。建议名称间的区别要显而易见。

规则 5.2 (强制)： 具有内部作用域的标识符不应使用与具有外部作用域的标识符相同的名称, 这会隐藏了外部标识符。

(5.2 Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.)

外部作用域和内部作用域的定义如下。文件范围内的标识符可以看做是具有最外部 (outermost) 的作用域; 块范围内的标识符看做是具有更内部 (more inner) 的作用域; 连续嵌套的块, 其作用域更深入。本规则只是不允许一个第二深层 (second inner) 的定义隐

藏其外层的定义，如果第二个定义没有隐藏第一个定义，那么就不算违反本规则。
在嵌套的范围中，使用相同名称的标识符隐藏其他标识符会使得代码非常混乱。例如：

```
int16_t i;
{
    int16_t i; /* 这是另一个不同的变量 */
    /* 它是不允许出现的 */
    i = 3; /* 这会引起混乱（主要指写程序或者读程序的人可能会弄乱） */
}
```

解决的方法很简单，就是不定义重复的变量。（想 for 循环中常用的 i、j、k 等除外）。

规则 5.3（强制）： typedef 的名字应当是唯一的标识符。

（5.3 A typedef name shall be a unique identifier）

typedef 的名称不能重用，不管是做为其他typedef 或者任何目的。例如：

```
{
    typedef unsigned char uint8_t;
}
{
    typedef unsigned char uint8_t; /* 不允许-重复定义 */
}
{
    unsigned char uint8_t; /* 不允许 -uint8_t被重复使用 */
}
```

typedef 的名称不能在程序中的任何地方重用。如果类型定义是在头文件中完成的，而该头文件被多个源文件包含，不算违背本规则。

规则 5.4（强制）： 标签（tag）名称必须是唯一的标识符。

（5.4 A tag name shall be a unique identifier.）

程序中标签的名字不可重用，不管是做为另外的标签还是出于其他目的。标签的所有使用或者用于结构类型标识符（struct），或者用于联合类型标识符（union），例如：

```
struct stag
{
    uint16_t a;
    uint16_t b;
};
struct stag a1 = { 0, 0 }; /* 允许 - compatible with above */
union stag a2 = { 0, 0 }; /* 不允许 - not compatible with previous declarations */
void foo (void)
{
    struct stag { uint16_t a; }; /* 不允许 - 标签 stag 重复定义 */
}
```

如果类型定义是在头文件中完成的，且头文件被多个源文件包含，那么规则不算违背。

规则 5.5（建议）： 具有静态存储期的对象或函数标识符不能重用。

（5.5 No object or function identifier with static storage duration should be reused.）

不管作用域如何，具有静态存储期的标识符都不应在系统内的所有源文件中重用。它包含带有外部链接的对象或函数，及带有静态存储类标识符的任何对象或函数。

由于编译器能够理解这一点而且决不会发生混淆，那么对用户来说就存在着把不相关的变量以相同名字联系起来的可能性。

这种混淆的例子之一是，在一个文件中存在一个具有内部链接的标识符，而在另外一个文件中存在着具有外部链接的相同名字的标识符。

规则 5.6（建议）： 一个命名空间中不应存在与另外一个命名空间中的标识符拼写相同的标识符，除了结构和联合中的成员名字。

(5.6 No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.)

下面给出了违背此规则的例子，其中 `value` 在不经意中代替了 `record.value`：

```
struct
{
    int16_t key ;
    int16_t value ;
} record ;
int16_t value; /* 违反规则 */
record.key = 1;
value = 0; /* 本应该是record.value */
```

相比之下，下面的例子没有违背此规则，因为两个成员名字不会引起混淆：

```
struct device_q
{
    struct device_q *next ;    /*定义指向本身的指针*/
}devices[N_DEVICES] ;
struct task_q
{
    struct task_q *next ;    /* 定义指向本身的指针 */
}tasks[N_TASKS];
device[0].next = &devices[1];
tasks[0].next = &tasks[1];
```

规则5.7（建议）： 不能重用标识符名字。

(5.7 No identifier name should be reused.)

不考虑作用域，系统内任何文件中不应重用标识符。

6 类型

规则 6.1（强制）： 单纯的 `char` 类型应该只用做存储和使用字符值。

(6.1 The plain char type shall be used only for the storage and use of character values.)

规则 6.2（强制）： `signed char` 和 `unsigned char` 类型应该只用做存储和使用数字值。

(6.2 Signed and unsigned char type shall be used only for the storage and use of numer

ic values.)

有三种不同的 `char` 类型：（单纯的）`char`、`unsigned char`、`signed char`。`unsigned char` 和 `signed char` 用于数字型数据，`char` 用于字符型数据。单纯`char` 类型的符号是实现定义的，不应依赖。单纯 `char` 类型所能接受的操作只有赋值和等于操作符（=、==、!=）。

规则 6.3（建议）： 对于基本的类型使用Typedef来表示大小和有无符号。

（6.3 Typedefs that indicate size and signedness should be used in place of the basic types.）

例：

```
typedef char char_t;
typedef signed char int8_t;
typedef signed short int16_t;
typedef signed int int32_t
```

嵌入式操作系统uCOS-ii则这样表示：

```
typedef unsigned char INT8U;
typedef signed char   INT8S;
typedef unsigned int  INT16U;
...
```

相比之下，我更喜欢：

```
typedef unsigned char uchar;
...
```

规则 6.4（强制）： 位域只能被定义为 `unsigned int` 或 `signed int` 类型。

（6.4 Bit fields shall only be defined to be of type unsigned int or signed int.）

因为`int` 类型的位域可以是`signed` 或`unsigned`，使用`int` 是由实现定义的。由于其行为未被定义，所以不允许为位域使用 `enum`、`short` 或 `char` 类型。

规则 6.5（强制）： `unsigned int` 类型的位域至少应该为2 bits 长度。

（6.5 Bit fields of type signed int shall be at least 2 bits long.）

1 bit 长度的有符号位域是无用的。

7 常量

规则 7.1（强制）： 不应使用八进制常量（零除外）和八进制`escape` 序列。

（7.1 Octal constants (other than zero) and octal escape sequences shall not be used.）

任何以“0”（零）开始的整型常量都被看做是八进制的，所以这是危险的，如在书写固定长度的常量时。例如，下面为3 个数字位的总线消息做数组初始化时将产生非预期的结果（052 是八进制的，即十进制的42）：

```
code[1] = 109; /*相当于十进制的109 */
code[2] = 100; /*相当于十进制的100 */
code[3] = 052; /*相当于十进制的42 */
code[4] = 071; /* 相当于十进制的57 */
```

8 声明与定义

规则 8.1（强制）： 函数应当具有原型声明，且原型在函数的定义和调用范围内都是可见的。

(8.1 Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.)

对外部函数来说，我们建议采用如下方法，在头文件中声明函数（亦即给出其原型），并在所有需要该函数原型的代码文件中包含这个头文件。

规则 8.2（强制）： 不论何时声明或定义了一个对象或函数，它的类型都应显式声明。

(8.2 Whenever an object or function is declared or defined, its type shall be explicitly stated.)

举例：

```
extern x;           /* 不允许- 隐式的int类型定义 */
extern int16_t x;   /* 允许 - 明确的类型 */
const y;           /* 不允许 - 隐式的int类型定义 */
const int16_t y;    /* Compliant - explicit type */
static foo(void);  /* 不允许- 隐式的类型 */
static int16_t foo(void); /* 允许 - 明确的类型 */
```

当然，局部自动变量是可以省略auto的。

规则8.3（强制）： 函数的每个参数类型在声明和定义中必须是等同的，函数的返回类型也该是等同的。

(8.3 For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.)

规则 8.4（强制）： 如果对象或函数被声明了多次，那么它们的类型应该是兼容的。

(8.4 If objects or functions are declared more than once their types shall be compatible.)

规则8.5（强制）： 头文件中不应有对象或函数的定义。

(8.5 There shall be no definitions of objects or functions in a header file.)

头文件应该用于声明对象、函数、typedef 和宏，而不应该包含或生成占据存储空间的对象或函数（或它们的片断）的定义。这样就清晰地划分了只有C 文件才包含可执行的源代码，而头文件只能包含声明。

规则 8.6（强制）： 函数应该声明为具有文件作用域。

(8.6 Functions shall be declared at file scope.)

在块作用域中声明函数会引起混淆并可能导致未定义的行为。

规则 8.7（强制）： 如果对象的访问只是在单一的函数中，那么对象应该在块范围内声明。

(8.7 Objects shall be defined at block scope if they are only accessed from within a single function.)

可能的情况下，对象的作用域应该限制在函数内。只有当对象需要具有内部或外部链接时才能为其使用文件作用域。当在文件范围内声明对象时，使用规则8.10。良好的编程实践是，在不必要的情况下避免使用全局标识符。对象声明在最外层或最内层的做法主要是种风

格问题。

规则 8.8（强制）： 外部对象或函数应该声明在唯一的文件中。

(8.8 An external object or function shall be declared in one and only one file.)

通常这意味着在一个头文件中声明一个外部标识符，而在定义或使用该标识符的任何文件中包含这个头文件。例如，

在头文件featureX.h 中声明：

```
extern int16_t a ;
```

然后在.c文件中对a 进行定义：

```
#include <featureX.h> //包含声明外部对象的头文件
```

```
int16_t a = 0 ;
```

工程中存在的头文件可能是一个或多个，但是任何一个外部对象或函数都只能在一个头文件中声明。

规则 8.9（强制）： 具有外部链接的标识符应该具有准确的外部定义。

(8.9 An identifier with external linkage shall have exactly one external definition.)

一个标识符如果存在多个定义（在不同的文件中）或者甚至没有定义，那么其行为是未定义的。不同文件中的多个定义是不允许的，即使这些定义相同也不允许；进而如果这些定义不同或者标识符的初始值不同，问题显然很严重。

规则 8.10（强制）： 在文件范围内声明和定义的所有对象或函数应该具有内部链接，除非是在需要外部链接的情况下。

(8.10 All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.)

如果一个函数只是在同一文件中的其他地方调用，那么就用static。使用static 存储类标识符将确保标识符只是在声明它的文件中是可见的，并且避免了和其他文件或库中的相同标识符发生混淆的可能性。

规则 8.11（强制）： static 存储类标识符应该用于具有内部链接的对象和函数的定义和声明。

(8.11 The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.)

static 和extern 存储类标识符常常是产生混淆的原因。良好的编程习惯是，把static 关键字一致地应用在所有具有内部链接的对象和函数的声明上。

规则 8.12（强制）： 当一个数组声明为具有外部链接，它的大小应该显式声明或者通过初始化进行隐式定义。

(8.12 When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.)

举例：

```
int array1[10] ; /* Compliant */
```

```
extern int array2[] ; /* Not compliant */
```

```
int array2[] = { 0, 10, 15 } ; /* Compliant */
```

尽管可以在数组声明不完善时访问其元素，然而仍然是在数组的大小可以显式确定的情况下，这样做才会更为安全。

9 初始化

规则 9.1（强制）： 所有自动变量在使用前都应被赋值。

(9.1 All automatic variables shall have been assigned a value before being used.)

本规则的意图是使所有变量在其被读之前已经写过了，除了声明中的初始化。注意，根据 ISO C 标准，具有静态存储期的变量缺省地被自动赋予零值，除非经过了显式的初始化。实际中，一些嵌入式环境没有实现这样的缺省行为。静态存储期是所有以static存储类形式声明的变量或具有外部链接的变量的共同属性，自动存储期变量通常不是自动初始化的。

规则 9.2（强制）： 应该使用大括号以指示和匹配数组和结构体的非零初始化构造。

(9.2 Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.)

ISO C 要求数组、结构体和联合体的初始化列表要以一对大括号括起来（尽管不这样做的行为是未定义的）。本规则更进一步地要求，使用附加的大括号来指示嵌套的结构。它迫使程序员显式地考虑和描述复杂数据类型元素（比如，多维数组）的初始化次序。

例如，下面的例子是二维数组初始化的有效（在 ISO C 中）形式，但第一个与本规则相违背：

```
int16_t y[3][2] = { 1, 2, 3, 4, 5, 6 }; /* 不允许 */
```

```
int16_t y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } }; /* 允许*/
```

在结构中以及在结构、数组和其他类型的嵌套组合中，规则类似。

还要注意的，数组或结构的元素可以通过只初始化其首元素的方式初始化（为0 或 NULL）。如果选择了这样的初始化方法，那么首元素应该被初始化为0（或NULL），此时不需要使用嵌套的大括号。

规则 9.3（强制）： 在枚举列表中，“=” 不能显式用于除首元素之外的元素上，除非所有的元素都是显式初始化的。

(9.3 In an enumerator list, the “=” construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly Initialised.)

如果枚举列表的成员没有显式地初始化，那么 C 将为其分配一个从0 开始的整数序列，首元素为0，后续元素依次加1。

如上规则允许的，首元素的显式初始化迫使整数的分配从这个给定的值开始。当采用这种方法时，重要的是确保所用初始化值一定要足够小，这样列表中的后续值就不会超出该枚举常量所用的int 存储量。

列表中所有项目的显式初始化也是允许的，它防止了易产生错误的自动与手动分配的混合。然而，程序员就该担负职责以保证所有值都处在要求的范围内以及值不是被无意复制的。

```
enum colour
{
    red = 3, blue, green, yellow = 5
}; /* 不允许 */
/* green 和yellow 代表相同的值 – 这是重复的*/
enum colour
{
```

```

    red = 3, blue = 4, green = 5, yellow = 5
}; /* 允许*/
/* green和 yellow 代表相同的值 -就让它们重复 */

```

10 数值类型转换

10.1 隐式和显式类型转换

C 语言给程序员提供了相当大的自由度并允许不同数值类型可以自动转换。由于某些功能性的原因可以引入显式的强制转换，例如：

- 用以改变类型使得后续的数值操作可以进行
- 用以截取数值
- 出于清晰的角度，用以执行显式的类型转换

为了代码清晰的目的而插入的强制转换通常是有用的，但如果过多使用就会导致程序的可读性下降。正如下面所描述的，一些隐式转换是可以安全地忽略的，而另一些则不能。

10.2 隐式转换的类型

存在三种隐式转换的类别需要加以区分。

1. 整数提升（Integral promotion）转换

整数提升描述了一个过程，借此过程，数值操作总是在int 或long（signed 或unsigned）整型操作数上进行。其他整型操作数（char、short、bit-field 和enum）在数值操作前总是先转化为int 或unsigned int 类型。这些类型称为small integer 类型。

整数提升的规则命令，在大多数数值操作中，如果 int 类型能够代表原来类型的所有值，那么small integer 类型的操作数要被转化为int 类型；否则就被转化为unsigned int。

注意，整数提升：

- 仅仅应用在 small integer 类型上
- 应用在一元、二元和三元操作数上
- 不能用在逻辑操作符（&&、||、!）的操作数上
- 用在 switch 语句的控制表达式上

整数提升经常和操作数的“平衡”（balancing，后面提到）发生混淆。事实上，整数提升发生在一元操作的过程中，如果二元操作的两个操作数是同样类型的，那么也可以发生在二元操作之上。

由于整数提升，两个类型为unsigned short 的对象相加的结果总是signed int 或unsigned int类型的；事实上，加法是在后面两种类型上执行的。因此对于这样的操作，就有可能获得一个其值超出了原始操作数类型大小的结果。例如，如果int 类型的大小是32 位，那么就能够把两个short（16 位）类型的对象相乘并获得一个32 位的结果，而没有溢出的危险。另一方面，如果int 类型仅是16 位，那么两个16 位对象的乘积将只能产生一个16 位的结果，同时必须对操作数的大小给出适当的限制。

整数提升还可以应用在一元操作符上。例如，对一个 unsigned char 操作数执行位非(~)运算，其结果典型地是signed int 类型的负值。

整数提升是 C 语言中本质上的不一致性，在这当中small integer 类型的行为与long 和int类型不同。MISRA-C 鼓励使用typedef。然而，由于众多整型的行为是不一致的，忽略基本类型（见后面的描述）可能是不安全的，除非对表达式的构造方式给出一些限制。后面规则的意图是想中和整数提升的后果以避免这些异常。

2. 赋值转换

赋值转换发生在：

- 赋值表达式的类型被转化成赋值对象的类型时
- 初始化表达式的类型被转化成初始化对象的类型时
- 函数调用参数的类型被转化成函数原型中声明的形式参数的类型时
- 返回语句中用到的表达式的类型被转化成函数原型中声明的函数类型时
- **switch-case** 标签中的常量表达式的类型被转化成控制表达式的提升类型时。这个转换仅用于比较的目的

每种情况中，必要时数值表达式的值是无条件转换到其他类型的。

3. 平衡转换 (Balancing conversions)

平衡转换的描述是在 ISO C 标准中的“Usual Arithmetic Conversions”条目下。这套规则提供一个机制，当二元操作符的两个操作数要平衡为一个通用类型时或三元操作符 (?:) 的第二、第三个操作数要平衡为一个通用类型时，产生一个通用类型。平衡转换总是涉及到两个不同类型的操作数；其中一个、有时是两个需要进行隐式转换。整数提升（上面描述的）的过程使得平衡转换规则变得复杂起来，在整数提升时，**small integer** 类型的操作数首先要提升到 **int** 或 **unsigned int** 类型。整数提升是常见的数值转换，即使两个操作数的类型一致。与平衡转换明显相关的操作符是：

- 乘除 *****、**/**、**%**
- 加减 **+**、**-**
- 位操作 **&**、**^**、**|**
- 条件操作符 **(... ? ... : ...)**
- 关系操作符 **>**、**>=**、**<**、**<=**
- 等值操作符 **==**、**!=**

其中大部分操作符产生的结果类型是由平衡过程产生的，除了关系和等值操作符，它们产生具有 **int** 类型的布尔值。

要注意的是，**位移操作符 (<<和>>)** 的操作数不进行平衡，运算结果被提升为第一个操作数的类型；第二个操作数可以是任何有符号或无符号的整型。

10.3 危险的类型转换

类型转换过程中存在大量潜在的危险需要加以避免：

- 数值的丢失：转化后的类型其数值量级不能被体现
- 符号的丢失：从有符号类型转换为无符号类型会导致符号的丢失
- 精度的丢失：从浮点类型转换为整型会导致精度的丢失

对于所有数据和所有可能的兼容性实现来说，唯一可以确保为安全的类型转换是：

- 整数值进行带符号的转换到更宽类型
- 浮点类型转换到更宽的浮点类型

当然，在实践中，如果假定了典型类型的大小，也能够把其他类型转换归类为安全的。普遍来说，**MISRA-C:2004** 采取的原则是，利用显式的转换来辨识潜藏的危险类型转换。类型转换中还有其他的一些危险需要认清。这些问题产生于 C 语言的难度和误解，而不是由于数据值不能保留。

□ 整数提升中的类型放宽：整数表达式运算的类型依赖于经过整数提升后的操作数的类型。总是能够把两个 8 位数据相乘并在有量级需要时访问 16 位的结果。有时而不总是能够把两个 16 位数相乘并得到一个 32 位结果。这是 C 语言中比较危险的不一致性，为了避免混淆，安全的做法是不要依赖由整数提升所提供的类型放宽。考虑如下例子：

```
uint16_t  u16a = 40000; /* unsigned short / unsigned int ? */
uint16_t  u16b = 30000; /* unsigned short / unsigned int ? */
uint32_t  u32x;         /* unsigned int / unsigned long ? */
```

```
u32x = u16a + u16b;    /* u32x = 70000 or 4464 ? */
```

期望的结果是 70000，但是赋给u32x的值在实际中依赖于int 实现的大小。如果int 实现的大小是32 位，那么加法就会在有符号的32 位数值上运算并且保存下正确的值。如果int 实现的大小仅是16 位，那么加法会在无符号的16 位数值上进行，于是会发生溢出

（wraparound）现象并产生值4464（70000%65536）。无符号数值的折叠（wraparound）是经过良好定义的甚至是有意的；但也会存在潜藏的混淆。

□ 类型计算的混淆：程序员中常见的概念混乱也会产生类似的问题，人们经常会以为参与运算的类型在某种方式上受到被赋值或转换的结果类型的影响。例如，在下面的代码中，两个16 位对象进行16 位的加法运算（除非被提升为32 位int），其结果在赋值时被转换为uint32_t 类型。

```
u32x = u16a + u16b;
```

并非少见的是，程序员会认为此表达式执行的是 32 位加法——因为u32x 的类型。对这种特性的混淆不只局限于整数运算或隐式转换，下面的例子描述了在某些语句中，结果是良好定义的但运算并不会按照程序员设想的那样进行。

```
u32a = (uint32_t) (u16a * u16b);
f64a = u16a / u16b ;
f32a = (float32_t) (u16a / u16b) ;
f64a = f32a + f32b ;
f64a = (float64_t) (f32a + f32b) ;
```

□ 数学运算中符号的改变：整数提升经常会导致两个无符号的操作数产生一个（signed）int类型的结果。比如，如果int 是32 位的，那么两个16 位无符号数的加法将产生一个有符号的32 位结果；而如果int 是16 位的，那么同样运算会产生一个无符号的16 位结果。

□ 位运算中符号的改变：当位运算符应用在无符号短整型时，整数提升会有某些特别不利的反响。比如，在一个unsigned char 类型的操作数上做位补运算通常会产生其值为负的（signed）int 类型结果。在运算之前，操作数被提升为int 类型，并且多出来的那些高位被补运算置1。那些多余位的个数，若有的话，依赖于int 的大小，而且在补运算后接右移运算是危险的。

为了避免上述问题产生的危险，重要的是要建立一些准则以限制构建表达式的方式。这里首先给出某些概念的定义。

10.4 基本类型（underlying type）

表达式的类型是指其运算结果的类型。当两个long 类型的值相加时，表达式具有long 类型。大多数数值运算符产生其类型依赖于操作数类型的结果。另一方面，某些操作符会给出具有int 类型的布尔结果而不管其操作数类型如何。所以，举例来说，当两个long 类型的项用关系运算符做比较时，该表达式的类型为int。

术语“基本类型”的定义是，在不考虑整数提升的作用下描述由计算表达式而得到的类型。当两个 int 类型的操作数相加时，结果是int 类型，那么表达式可以说具有int 类型。当两个 unsigned char 类型的数相加时，结果也是int 类型（通常如此，因为整数提升的原因），但是该表达式基本的类型按照定义则是unsigned char。

术语“基本类型”不是 C 语言标准或其他C 语言的文本所认知的，但在描述如下规则时它很有用。它描述了一个假想的对C 语言的违背，其中不存在整数提升而且常用的数值转换一致性地应用于所有的整数类型。引进这样的概念是因为整数提升很敏感且有时是危险的。整数提升是C 语言中不可避免的特性，但是这些规则的意图是要使整数提升的作用能够通过不利用发生在small integer 操作数上的宽度扩展来中和。当然，C 标准没有显式地定义在缺乏整数提升时small integer 类型如何平衡为通用类型，尽管标准确实建立了值保

留（value-perserving）原则。

当int 类型的数相加时，程序员必须要确保运算结果不会超出int 类型所能体现的值。如果他没有这样做，就可能会发生溢出而结果值是未定义的。这里描述的方法意欲使得在做small integer 类型的加法时使用同样的原则；程序员要确保两个unsigned char 类型的数相加的结果是能够被unsigned char 体现的，即使整数提升会引起更大类型的计算。换句话说，对表达式基本类型的遵守要多于其真实类型。

整数常量表达式的基本类型

C 语言的一个不利方面是，它不能定义一个char 或short 类型的整型常量。比如，值“5”可以通过附加的一个合适的后缀来表示为int、unsigned int、long 或unsigned long 类型的常量；

但没有合适的后缀用来创建不同的char 或short 类型的常量形式。这为维护表达式中的类型一致性提出了困难。如果需要为一个unsigned char 类型的对象赋值，那么或者要承受对一个整数类型的隐式转换，或者要实行强制转换。很多人会辩称在这种情况下使用强制转换只能导致可读性下降。

在初始化、函数参数或数值表达式中需要常量时也会遇到同样的问题。然而只要遵守强类型（strong typing）原则，这个问题就会比较乐观。

解决该问题的一个方法是，想象整型常量、枚举常量、字符常量或者整型常量表达式具有适合其量级的类型。这个目标可以通过以下方法达到，即延伸基本类型的概念到整型常量上，并想象在可能的情况下数值常量已经通过提升想象中的具有较小基本类型的常量而获得。

这样，整型常量表达式的基本类型就可以如下定义：

1. 如果表达式的真实类型是（signed）int，其基本类型就是能够体现其值的最小的有符号整型。
2. 如果表达式的真实类型是 unsigned int，其基本类型就是能够体现其值的最小的无符号整型。
3. 在所有其他情况下，表达式的基本类型与其真实类型相同。

在常规的体系结构中，整型常量表达式的基本类型可以根据其量级和符号确定如下：

无符号值：

0U	to	255U	8 bit unsigned
256U	to	65535U	16 bit unsigned
65536U	to	4294967295U	32 bit unsigned

有符号值：

-2147483648	to	-32769	32 bit signed
-32768	to	-129	16 bit signed
-128	to	127	8 bit signed
128	to	32767	16 bit signed
32768	to	2147483647	32 bit signed

10.5 复杂表达式（complex expressions）

后面章节中描述的类型转换规则在某些地方是针对“复杂表达式”的概念。术语“复杂表达式”意味着任何不是如下类型的表达式：

- 非常量表达式
- lvalue（即一个对象）
- 函数的返回值

应用在复杂表达式的转换也要加以限制以避免上面总结出的危险。特别地，表达式中的数值运算序列需要以相同类型进行。

以下是复杂表达式：

```
s8a + s8b
~u16a
u16a >> 2
foo(2) + u8a
*ppc + 1
++u8a
```

以下不是复杂表达式，尽管某些包含了复杂的子表达式：

```
pc[u8a]
foo(u8a + u8b)
++ppuc
** (ppc + 1)
pcbbuf[s16a * 2]
```

10 隐式类型转换

规则 10.1（强制）： 下列条件成立时，整型表达式的值不应隐式转换为不同的基本类型：

- a) 转换不是带符号的向更宽整数类型的转换
- b) 表达式是复杂表达式
- c) 表达式不是常量而是函数参数
- d) 表达式不是常量而是返回的表达式。

(10.1 The value of an expression of integer type shall not be implicitly converted to a different underlying type if:

- a) it is not a conversion to a wider integer type of the same signedness, or
- b) the expression is complex, or
- c) the expression is not constant and is a function argument, or
- d) the expression is not constant and is a return expression)

规则 10.2（强制）： 下列条件成立时，浮点类型表达式的值不应隐式转换为不同的类型：

- a) 转换不是向更宽浮点类型的转换，或者
- b) 表达式是复杂表达式，或者
- c) 表达式是函数参数，或者
- d) 表达式是返回表达式。

(10.2 The value of an expression of floating type shall not be implicitly converted to a different type if:

- a) it is not a conversion to a wider floating type, or
- b) the expression is complex, or
- c) the expression is a function argument, or
- d) the expression is a return expression)

还要注意，在描述整型转换时，始终关注的是基本类型而非真实类型。

这两个规则广泛地封装了下列原则：

- 有符号和无符号之间没有隐式转换
- 整型和浮点类型之间没有隐式转换
- 没有从宽类型向窄类型的隐式转换
- 函数参数没有隐式转换
- 函数的返回表达式没有隐式转换
- 复杂表达式没有隐式转换

限制复杂表达式的隐式转换的目的，是为了要求在一个表达式里的数值运算序列中，所有的运算应该准确地以相同的数值类型进行。注意这并不是说表达式中的所有操作数必须具备相同的类型。

表达式 `u32a + u16b + u16c` 是合适的——两个加法在概念上（**notionally**）都以U32 类型进行

表达式 `u16a + u16b + u32c` 是不合适的——第一个加法在概念上以U16 类型进行，第二个加法是U32 类型的。（注：`u16a + u16b + u32c`相当于 `(u16a + u16b) + u32c`）

使用名词“在概念上”是因为，在实际中数值运算的类型将依赖于int 实现的大小（与微处理器的构架，编译器的位数有关）。通过遵循这样的原则，所有运算都以一致的（基本）类型来进行，能够避免程序员产生的混淆和与整数提升有关的某些危险。

（以下u8a表示8位无符号整数，s8a表示8位有符号整数。compliant表示允许）

```
extern void foo1 (uint8_t x);
int16_t t1 (void)
{
    ...
    foo1 (u8a); /* compliant */
    foo1 (u8a + u8b); /* compliant */
    foo1 (s8a); /* not compliant */
    foo1 (u16a); /* not compliant */
    foo1 (2); /* not compliant */
    foo1 (2U); /* compliant */
    foo1 ( (uint8_t) 2 ); /* compliant */
    ... s8a + u8a /* not compliant */
    ... s8a + (int8_t) u8a /* compliant */
    s8b = u8a; /* not compliant */
    ... u8a + 5 /* not compliant */
    ... u8a + 5U /* compliant */
    ... u8a + (uint8_t) 5 /* compliant */
    u8a = u16a; /* not compliant */
    u8a = (uint8_t) u16a; /* compliant */
    u8a = 5UL; /* not compliant */
    ... u8a + 10UL /* compliant */
    u8a = 5U; /* compliant */
    ... u8a + 3 /* not compliant */
    ... u8a >> 3 /* compliant */
    ... u8a >> 3U /* compliant */
    pca = "P"; /* compliant */
}
```

```

... s32a + 80000 /* compliant */
... s32a + 80000L /* compliant */
f32a = f64a; /* not compliant */
f32a = 2.5; /* not compliant –
unsuffixed floating
constants are of type
double */
u8a = u8b + u8c; /* compliant */
s16a = u8b + u8b; /* not compliant */
s32a = u8b + u8c; /* not compliant */
f32a = 2.5F; /* compliant */
u8a = f32a; /* not compliant */
s32a = 1.0; /* not compliant */
f32a = 1; /* not compliant */
f32a = s16a; /* not compliant */
... f32a + 1 /* not compliant */
... f64a * s32a /* not compliant */
...
return (s32a); /* not compliant */
...
return (s16a); /* compliant */
...
return (20000); /* compliant */
...
return (20000L); /* not compliant */
...
return (s8a); /* not compliant */
...
return (u16a); /* not compliant */
}
int16_t foo2 (void)
{
...
... (u16a + u16b) + u32a /* not compliant */
... s32a + s8a + s8b /* compliant */
... s8a + s8b + s32a /* not compliant */
f64a = f32a + f32b; /* not compliant */
f64a = f64b + f32a; /* compliant */
f64a = s32a / s32b; /* not compliant */
u32a = u16a + u16a; /* not compliant */
s16a = s8a; /* compliant */
s16a = s16b + 20000; /* compliant */
s32a = s16a + 20000; /* not compliant */
s32a = s16a + (int32_t) 20000; /* compliant */

```

```

u16a = u16b + u8a; /* compliant */
foo1 (u16a); /* not compliant */
foo1 (u8a + u8b); /* compliant */
...
return s16a; /* compliant */
...
return s8a; /* not compliant */
}

```

10.2 显式转换（强制转换）

规则 10.3（强制）： 整型复杂表达式的值只能强制转换到更窄的类型且与表达式的基本类型具有相同的符号。

(10.3 The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.)

规则 10.4（强制）： 浮点类型复杂表达式的值只能强制转换到更窄的浮点类型。

(10.4 The value of a complex expression of floating type may only be cast to a narrower floating type.)

如果强制转换要用在任何复杂表达式上，可以应用的转换的类型应该严格限制。复杂表达式的转换经常是混淆的来源，保持谨慎是明智的做法。为了符合这些规则，有必要使用临时变量并引进附加的语句。

```

(float32_t) (f64a + f64b) /* compliant */
(float64_t) (f32a + f32b) /* not compliant */
... (float64_t) f32a /* compliant */
... (float64_t) (s32a / s32b) /* not compliant */
... (float64_t) (s32a > s32b) /* not compliant */
... (float64_t) s32a / (float32_t) s32b /* compliant */
... (uint32_t) (u16a + u16b) /* not compliant */
... (uint32_t) u16a + u16b /* compliant */
... (uint32_t) u16a + (uint32_t) u16b /* compliant */
... (int16_t) (s32a - 12345) /* compliant */
... (uint8_t) (u16a * u16b) /* compliant */
... (uint16_t) (u8a * u8b) /* not compliant */
... (int16_t) (s32a * s32b) /* compliant */
... (int32_t) (s16a * s16b) /* not compliant */
... (uint16_t) (f64a + f64b) /* not compliant */
... (float32_t) (u16a + u16b) /* not compliant */
... (float64_t) foo1 (u16a + u16b) /* compliant */
... (int32_t) buf16a[u16a + u16b] /* compliant */

```

规则10.5（强制）： 如果位运算符 `~` 和 `<<` 应用在基本类型为 `unsigned char` 或 `unsigned short` 的操作数，结果应该立即强制转换为操作数的基本类型。

(10.5 If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.)

当这些操作符（`~`和`<<`）用在 `small integer` 类型（`unsigned char` 或 `unsigned short`）时，

运算之前要先进行整数提升，结果可能包含并非预期的高端数据位。例如：

```
uint8_t  port = 0x5aU;
uint8_t  result_8;
uint16_t result_16;
uint16_t mode;
result_8 = (~port) >> 4; /* not compliant */
```

~port 的值在16 位机器上是0xffa5，而在32 位机器上是0xfffffa5。在这种情况下，result_8的值是0xfa，然而期望值可能是0x0a。这样的危险可以通过如下所示的强制转换来避免：

```
result_8 = ( ( uint8_t ) (~port ) ) >> 4; /* compliant */
result_16 = ( ( uint16_t ) (~(uint16_t) port ) ) >> 4; /* compliant */
```

当<<操作符用在small integer 类型时会遇到类似的问题，高端数据位被保留下来。例如：

```
result_16 = ( ( port << 4 ) & mode ) >> 6; /* not compliant */
```

result_16 的值将依赖于int 实现的大小。附加的强制转换可以避免任何模糊性。

```
result_16 = ( ( uint16_t ) ( ( uint16_t ) port << 4 ) & mode ) >> 6; /* compliant */
```

10.3 整数后缀

规则 10.6（强制）： 后缀“U”应该用在所有unsigned 类型的常量上。

（10.6 A “U” suffix shall be applied to all constants of unsigned type）

整型常量的类型是混淆的潜在来源，因为它依赖于许多因素的复杂组合，包括：

- 常数的量级
- 整数类型实现的大小
- 任何后缀的存在
- 数值表达的进制（即十进制、八进制或十六进制）

例如，整型常量“40000”在32 位环境中是int 类型，而在16 位环境中则是long 类型。值0x8000 在16 位环境中是unsigned int 类型，而在32 位环境中则是（signed）int 类型。

注意：

- 任何带有“U”后缀的值是unsigned 类型
- 一个不带后缀的小于 2的31次方的十进制值是signed 类型

但是：

- 不带后缀的大于或等于 2的15次方的十六进制数可能是signed 或unsigned 类型
- 不带后缀的大于或等于 2的31次方的十进制数可能是signed 或unsigned 类型

常量的符号应该明确。符号的一致性构建良好形式的表达式的重要原则。如果一个常数是unsigned 类型，为其加上“U”后缀将有助于避免混淆。当用在较大数值上时，后缀也许也是多余的（在某种意义上它不会影响常量的类型）；然而后缀的存在对代码的清晰性是种有价值的帮助。

6.11 指针类型转换

指针类型可以归为如下几类：

- 对象指针
- 函数指针
- void 指针

- 空 (null) 指针常量 (即由数值0 强制转换为void*类型)

涉及指针类型的转换需要明确的强制，除非在以下时刻：

- 转换发生在对象指针和 void 指针之间，而且目标类型承载了源类型的所有类型标识符
 - 当空指针常量 (void*) 被赋值给任何类型的指针或与其做等值比较时，空指针常量被自动转化为特定的指针类型
- C 当中只定义了一些特定的指针类型转换，而一些转换的行为是实现定义的。

规则 11.1 (强制)：指针不能转换为函数或者整型以外的其他类型。

(11.1 Conversions shall not be performed between a pointer to a function and any type other than an integral type.)

函数指针到不同类型指针的转换会导致未定义的行为。举个例子，这意味着一个函数指针不能转换成指向不同类型函数的指针。

规则 11.2 (强制)：对象指针和其他除整型之外的任何类型指针之间、对象指针和其他类型对象的指针之间、对象指针和void 指针之间不能进行转换。

(11.2 Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.)

这些转换未经定义。

规则 11.3 (建议)：不应在指针类型和整型之间进行强制转换

(11.3 A cast should not be performed between a pointer type and an integral type.)

当指针转换到整型时所需要的整型的大小是实现定义的。尽可能的情况下要避免指针和整型之间的转换，但是在访问内存映射寄存器或其他硬件特性时这是不可避免的。

规则 11.4 (建议)：不应在某类型对象指针和其他不同类型对象指针之间进行强制转换。

(11.4 A cast should not be performed between a pointer to object type and a different pointer to object type.)

如果新的指针类型需要更严格的分配时这样的转换可能是无效的。

```
uint8_t * p1;
uint32_t * p2;
p2 = (uint32_t *) p1; /* 不相匹配的转换 ? */
```

规则11.5 (强制)：如果指针所指向的类型带有const 或volatile 限定符，那么移除限定符的强制转换是不允许的。

(11.5 A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.)

任何通过强制转换移除类型限定符的企图都是对类型限定符规则的违背。注意，这里所指的限定符与任何可以应用在指针本身的限定符不同。

```
uint16_t x;
uint16_t * const cpi = &x; /* 常量指针*/
uint16_t * const * pcpi ; /* 指向常量指针的指针*/
const uint16_t ** ppci ; /* 指向指针的常量指针*/
uint16_t ** ppi;
const uint16_t * pci; /* 指向常量的指针t */
```

```
volatile uint16_t * pvi; /* volatile类型指针 */
uint16_t * pi;
...
pi = cpi; /* 允许 – no conversion no cast required */
pi = (uint16_t *)pci; /* 不允许 */
pi = (uint16_t *)pvi; /* 不允许 */
ppi = (uint16_t *)pcpi; /* 不允许 */
ppi = (uint16_t *)ppci; /* 不允许 */
```

12 表达式

规则 12.1 (建议)： 不要过分依赖C 表达式中的运算符优先规则

(12.1 Limited dependence should be placed on C's operator precedence rules in expressions.)

括号的使用除了可以覆盖缺省的运算符优先级以外，还可以用来强调所使用的运算符。使用相当复杂的C 运算符优先级规则很容易引起错误，那么这种方法就可以帮助避免这样的错误，并且可以使得代码更为清晰可读。然而，过多的括号会分散代码使其降低了可读性。下面的方针给出了何时使用括号的建议：

- 赋值运算符的右手操作数不需要使用括号，除非右手端本身包含了赋值表达式：

```
x = a + b; /* 允许 */
x = (a + b); /* ( ) 是不需要的 */
```

- 一元运算符的操作数不需要使用括号：

```
x = a * -1; /* 允许 */
x = a * (-1); /* ( ) 是不需要的 */
```

- 否则，二元和三元运算符的操作数应该是cast-expressions，除非表达式中所有运算符是相同的。

```
x = a + b + c; /* 允许，但要求a、b、c的数据类型相同 */
x = f(a + b, c); /* no ( ) required for a + b */
x = (a == b) ? a : (a - b);
if (a && b && c) /* acceptable */
x = (a + b) - (c + d);
x = (a * 3) + c + d;
x = (uint16_t) a + b; /* no need for ( (uint16_t) a) */
```

- 即使所有运算符都是相同的，也可以使用括号控制运算的次序。某些运算符（如，加法和乘法）在代数学上结合律的，而在C 中未必如此。类似地，涉及混合类型的整数运算（许多规则不允许）因为整数提升的存在可以产生不同的结果。下面的例子是按照16 位的实现写成的，它描述了加法不是结合的以及表达式结构清晰的重要性：

```
uint16_t a = 10;
uint16_t b = 65535;
uint32_t c = 0;
uint32_t d;
d = (a + b) + c; /* d 等于9; a + b 的值会溢出（16位最大65536） */
d = a + (b + c); /* d 等于 65545 */
/* this example also deviates from several other rules */
```

注意，规则12.5 是本规则的特例，它只能应用在逻辑运算符（&& 和 ||）上。

规则 12.2（强制）：表达式的值应和标准允许的评估顺序一致。

（12.2 The value of an expression shall be the same under any order of evaluation that the standard permits.）

除了少数运算符（特别地，函数调用运算符（）、&&、||、?: 和 ,（逗号））之外，子表达式所依据的运算次序是未指定的并会随时更改。这意味着不能信任子表达式的运算次序，特别不能信任可能会发生副作用（side effect）的运算次序。在表达式运算中的某些点上，如果能保证所有先前的副作用都已经发生，那么这些点称为“序列点（sequence point）”。序列点和副作用的描述见ISO 9899:1990的5.1.2.3 节、6.3 节和6.6 节。

注意，运算次序的问题不能使用括号来解决，因为这不是优先级的問題。

下面的条款告诉我们对运算次序的依赖是如何发生的，并由此帮助我们采纳本规则。

1. 自增或自减运算符

做为能产生错误的例子，考虑

```
x = b[i] + i++;
```

根据 b[i] 的运算是先于还是后于 i++ 的运算，表达式会产生不同的结果。把增值运算做为单独的语句，可以避免这个问题。那么：

```
x = b[i] + i;  
i ++;
```

2. 函数参数

函数参数的运算次序是未指定的。

```
x = func ( i++, i);
```

根据函数的两个参数的运算次序不同，表达式会给出不同的结果。

3. 函数指针

如果函数是通过函数指针调用的，那么函数标识符和函数参数运算次序是不可信任的。

```
p->task_start_fn (p++);
```

4. 函数调用

函数在被调用时可以具有附加的作用（如，修改某些全局数据）。可以通过在使用函数的表达式之前调用函数并为值采用临时变量的方法避免对运算次序的依赖。

例如

```
x = f (a) + g (a);
```

可以写成

```
x = f (a);  
x += g (a);
```

做为可以产生错误的例子，考虑下面的表达式，它从堆栈中取出两个值，从第一个值中减去第二个值，再把结果放回栈中：

```
push ( pop () - pop () );
```

根据哪一个 pop () 函数先进行计算（因为pop()具有副作用）会产生不同的结果。

5. 嵌套的赋值语句

表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。

例如，下面的做法是不赞成的：

```
x = y = y = z / 3;  
x = y = y++;
```

6.volatile 访问

类型限定符 `volatile` 是C 提供的，用来表示那些其值可以独立于程序的运行而自由更改的对象（例如输入寄存器）。对带有`volatile` 限定类型的对象的访问可能改变它的值。C 编译器不会优化对`volatile` 的读取，而且，据C 程序所关心的，对`volatile` 的读取具有副作用（改变`volatile` 的值）。做为表达式的一部分通常需要访问 `volatile` 数据，这意味着对运算次序的依赖。建议对`volatile` 的访问尽可能地放在简单的赋值语句中，如：

```
volatile uint16_t v;
/* ... */
x = v;
```

本规则讨论了带有副作用的运算次序问题。要注意子表达式的运算次数同样会带来问题，本规则没有提及。这是函数调用的问题，其中函数是以宏实现的。例如，考虑下面的函数宏及其调用：

```
#define MAX (a, b) ((a) > (b)) ? (a) : (b)
/* ... */
z = MAX (i++, j);
```

当 $a > b$ 时，该定义计算了两次第一个参数而在 $a \leq b$ 时只计算了一次。这样，宏调用根据 `i` 和 `j` 的值，对 `i` 增加了一次或两次。

应该说明的是，比如那些由浮点的四舍五入引起的量级依赖（magnitude-dependent）的作用也没有在这里涉及。尽管可能发生副作用的运算次序是未定义的，运算结果在另一方面是良好定义的并被表达式的结构所控制。在下面的例子中，`f1` 和`f2` 是浮点变量；`F3`、`F4` 和`F5`代表浮点类型的表达式。

```
f1 = F3 + (F4 + F5);
f2 = (F3 + F4) + F5;
```

加法运算的次序由括号的位置决定，至少表面如此。即，首先`F4` 的值加上`F5` 然后加上`F3`，给出`f1` 的值。假定`F3`、`F4` 和`F5` 没有副作用，那么它们的值独立于它们被计算的次序。然而，赋给`f1` 和`f2` 的值不能保证是相同的，因为浮点的四舍五入后紧接加法的运算将依赖于被加的值。

规则 12.3（强制）： 不能在具有副作用的表达式中使用`sizeof` 运算符。

(12.3 The `sizeof` operator shall not be used on expressions that contain side effects.)

C 当中存在的一个可能的编程错误，是为一个表达式使用了`sizeof` 运算符并期望计算表达式。然而表达式是不会被计算的：`sizeof` 只对表达式的类型有用。为避免这样的错误，`sizeof`不能用在具有副作用的表达式中，因为此时其副作用不会发生。例如：

```
int32_t i;
int32_t j;
j = sizeof (i = 1234);
/*表达式并没有执行，只是得到表达式类型int的size */
```

规则12.4（强制）： 逻辑运算符 `&&` 或 `||` 的右手操作数不能包含副作用。

(12.4 The right hand operand of a logical `&&` or `||` operator shall not contain side effects.)

C 当中存在这样的情况，表达式的某些部分不会被计算到。如果这些子表达式具有副作用，那么副作用可能会发生也可能不会发生，这依赖于其他子表达式的值。可以导致这种

问题的运算符是 `&&`、`||` 和 `?:`。前两种情况（逻辑运算符）下，右手操作数的计算是有条件的，依赖于左手操作数的值。在 `?:` 运算符情况下，或者第二个操作数被计算，或者第三个操作数被计算，却不会两者都被计算。两种逻辑运算符之一中，右手操作数的条件计算能轻易导致问题出现，如果程序员依赖副作用的发生。`?:` 运算符是被特殊用以在两个子表达式之间进行选择，因此导致错误的可能性较小。

例如：

```
if ( ishigh && ( x == i++ ) ) /* 不允许, i++可能不会被执行*/
if ( ishigh && ( x == f(x) ) ) /* Only acceptable if f(x) is known to have no side
effects */
```

可以产生副作用的运算在ISO 9899:1990中描述成volatile 对象的访问、对象的修改、文件的修改或是执行某些运算的函数的调用，这里，函数所执行的这些运算可以导致函数所运行的环境状态的改变。

规则 12.5（强制）： 逻辑 `&&` 或 `||` 的操作数应该是基本表达式（primary-expressions）。

（12.5 The operands of a logical `&&` or `||` shall be primary-expressions.）

“Primary expressions” 定义在ISO 9899:1990中。本质上它们或是单一的标识符，或是常量，或是括号括起来的表达式。本规则的作用是要求，如果操作数不是单一的标识符或常量，那么它必须被括起来。在这种情况下，括号对于代码的可读性和确保预期的行为都是非常重要的。如果表达式只由逻辑 `&&` 序列组成或逻辑 `||` 序列组成，就不需要使用括号。

例如：

```
if ( ( x == 0 ) && ishigh ) /* 突出x == 0 */
if ( x || y || z ) /* exception allowed, if x, y and z are Boolean */
if ( x || ( y && z ) ) /*突出y && z */
if ( x && ( !y ) ) /* make !y primary */
if ( ( is_odd(y) ) && x ) /* make call primary */
```

如果表达式只由逻辑 `&&` 序列组成或逻辑 `||` 序列组成，就不需要使用括号。

```
if ( ( x > c1 ) && ( y > c2 ) && ( z > c3 ) ) /* 允许*/
if ( ( x > c1 ) && ( y > c2 ) || ( z > c3 ) ) /* 不被允许, 不是相同序列 */
if ( ( x > c1 ) && ( ( y > c2 ) || ( z > c3 ) ) ) /* 允许临时使用 ( ) */
```

注意，本规则是规则12.1 的特例。

规则 12.6（建议）： 逻辑运算符（`&&`、`||` 和 `!`）的操作数应该是有效的布尔数。

（12.6 The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).）

有效布尔类型的表达式不能用做非逻辑运算符（`&&`、`||` 和 `!`）的操作数逻辑运算符 `&&`、`||` 和 `!` 很容易同位运算符 `&`、`|` 和 `~` 混淆。

规则 12.7（强制）： 位运算符不能用于基本类型（underlying type）是有符号的操作数上。

（12.7 Bitwise operators shall not be applied to operands whose underlying type is signed.）

位运算（`~`、`<<`、`>>`、`&`、`^` 和 `|`）对有符号整数通常是无意义的。比如，如果右移运算把符号位移动到数据位上或者左移运算把数据位移动到符号位上，就会产生问题。

规则 12.8（强制）： 移位运算符的右手操作数应该位于零和某数之间，这个数要小于左手

操作数的基本类型的位宽。

(12.8 The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.)

例如，如果左移或右移运算的左手操作数是16 位整型，那么要确保它移动的位数位于0和15 之间。

有多种确保遵循本规则的方法。对右手操作数来说，最简单的是使其为一个常数（其值可以静态检查）。使用无符号整型可以保证该操作数非负，那么只有其上限需要检查（在运行时动态检查或者通过代码复查）。否则这两种限制都要被检查。

```
u8a = (uint8_t) (u8a << 7); /* 允许 */
u8a = (uint8_t) (u8a << 9); /* 不允许，超出8位界限*/
u16a = (uint16_t) ((uint16_t) u8a << 9); /* 允许 */
```

规则12.9（强制）： 一元减运算符不能用在基本类型无符号的表达式上。

(12.9 The unary minus operator shall not be applied to an expression whose underlying type is unsigned.)

把一元减运算符用在基本类型为unsigned int 或unsigned long 的表达式上时，会分别产生类型为unsigned int 或unsigned long 的结果，这是无意义的操作。把一元减运算符用在无符号短整型的操作数上，根据整数提升的作用它可以产生有意义的有符号结果，但这不是好的方法。

规则 12.10（强制）： 不要使用逗号运算符。

(12.10 The comma operator shall not be used.)

使用逗号运算符通常不利于代码的可读性，可以使用其他方法达到相同的效果。

规则 12.11（建议）： 无符号整型常量表达式的计算不应产生折叠（wrap-around）。

(12.11 Evaluation of constant unsigned integer expressions should not lead to wrap-around.)

因为无符号整型表达式不会严格意义上地溢出，但会以模的方式产生折叠，因此任何无符号整型常量表达式的有效“溢出”将不会被编译器检测到。尽管在运行时（run-time）有很好的理由依赖于无符号整型提供的模运算，但在编译时（compile-time）计算常量表达式该理由就不那么明显了。因此任何发生折叠的无符号整型常量表达式都可能表示编程错误。本规则同等地应用于翻译过程的所有阶段。对于在编译时计算所选择的常量表达式，编译器以这样的方式计算，即其计算结果与在目标机上的运算结果相同，除了条件预处理指令。对这样的指令，可以使用常见的算术运算法则，但是int 和unsigned int 的行为会被分别替代成好像它们是long 或unsigned long 一样。

例如，在 int 类型为16 位、long 类型为32 位的机器上：

```
#define START 0x8000
#define END 0xFFFF
#define LEN 0x8000
#if ( (START + LEN) > END )
#error Buffer Overrun /* 可以，因为START和LEN会被认为是无符号long型 */
#endif
#if ( ( (END - START) - LEN) < 0 )
#error Buffer Overrun
```

```

/* 不可以: 相减后结果为 0xFFFFFFFF */
#endif
/* 对比上面和下面的START + LEN */
if ( ( START + LEN ) > END )
{
    error ( "Buffer overrun " );
/* 不可以: START + LEN 的结果是0x0000 (发生溢出) , 因为这是无符号int型
算术运算*/
}

```

规则12.12 (强制): 不应使用浮点数的基本 (underlying) 的位表示法 (bit representation) (12.12 The underlying bit representations of floating-point values shall not be used.)

浮点数的存储方法可以根据编译器的不同而不同, 因此不应使用直接依赖于存储方法的浮点操作。应该使用内置 (in-built) 的运算符和函数, 它们对程序员隐藏了存储细节。

规则 12.13 (建议): 在一个表达式中, 自增 (++) 和自减 (--) 运算符不应同其他运算符混合在一起。

(12.13 The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.)

不建议使用同其他算术运算符混合在一起的自增和自减运算符, 是因为:

- 它显著削弱了代码的可读性
 - 它为语句引入了其他的副作用, 可能存在未定义的行为
- 把这些操作同其他算术操作隔离开是比较安全的。

例如, 下面的语句是不适合的:

```
u8a = ++u8b + u8c--; /* 不允许 */
```

下面的序列更为清晰和安全:

```

++u8b;
u8a = u8b + u8c;
u8c --;

```

13 控制语句表达式

规则 13.1 (强制): 赋值运算符不能使用在产生布尔值的表达式上。

(13.1 Assignment operators shall not be used in expressions that yield a Boolean value.)

任何被认为是具有布尔值的表达式上都不能使用赋值运算。这排除了赋值运算符的简单与复杂的使用形式, 其操作数是具有布尔值的表达式。然而, 它不排除把布尔值赋给变量的操作。

如果布尔值表达式需要赋值操作, 那么赋值操作必须在操作数之外分别进行。这可以帮助避免 “=” 和 “==” 的混淆, 帮助我们静态地检查错误。

例如:

```

x = y;
if (x != 0)
{
    foo ();
}

```

```

    }
不能写成:
    if ((x = y) != 0) /* 产生布尔值 */
    {
        foo ();
    }
或者更坏的:
    if (x = y)
    {
        foo ();
    }

```

规则13.2（建议）： 数的非零检测应该明确给出，除非操作数是有效的布尔类型。

(13.2 Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.)

当要检测一个数据不等于零时，该测试要明确给出。本规则的例外是该数据代表布尔类型的值，虽然在C 中布尔数实际上也是整数。本规则的着眼点是在代码的清晰上，给出整数和逻辑数之间的清晰划分。

例如，如果x 是个整数，那么：

```

    if (x != 0) /* 测试x非零的恰当方法 */
    if (y) /* 不允许，除了有效的布尔类型以外 */

```

规则 13.3（强制）： 浮点表达式不能做相等或不等的检测。

(13.3 Floating-point expressions shall not be tested for equality or inequality.)

这是浮点类型的固有特性，等值比较通常不会计算为true，即使期望如此。而且，这种比较行为不能在执行前做出预测，它会随着实现的改变而改变。例如，下面代码中的测试结果就是不可预期的：

```

float32_t x, y;
/* some calculations in here */
if (x == y) /* not compliant */
{ /* ... */ }
if (x == 0.0f) /* not compliant */

```

间接的检测同样是有问题的，在本规则内也是禁止的。例如：

```

    if ((x <= y) && (x >= y))
    { /* ... */ }

```

为了获得确定的浮点比较，建议写一个实现比较运算的库。这个库应该考虑浮点的粒度（FLT_EPSILON）以及参与比较的数的量级。

规则 13.4（强制）： for 语句的控制表达式不能包含任何浮点类型的对象。

(13.4 The controlling expression of a for statement shall not contain any objects of floating type.)

控制表达式可能会包含一个循环计数器，检测其值以决定循环的终止。浮点变量不能用于此目的。舍入误差和截取误差会通过循环的迭代过程传播，导致循环变量的显著误差，并且在进行检测时很可能给出不可预期的结果。例如，循环执行的次数可以随着实现的改变而

改变，也是不可预测的。

规则 13.5（强制）： **for** 语句的三个表达式应该只关注循环控制。

(13.5 The three expressions of a for statement shall be concerned only with loop control.)

for 语句的三个表达式都给出时它们应该只用于如下目的：

第一个表达式初始化循环计数器（例子中的*i*）

第二个表达式应该包含对循环计数器（*i*）和其他可选的循环控制变量的测试

第三个表达式循环计数器（*i*）的递增或递减

规则 13.6（强制）： **for** 循环中用于迭代计数的数值变量不应在循环体中修改。

(13.6 Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.)

不能在循环体中修改循环计数器。然而可以修改表现为逻辑数值的其他循环控制变量。例如，指示某些事情已经完成的标记，然后在**for** 语句中测试。

```
flag = 1;
for ( i = 0; ( i < 5 ) && (flag == 1 ); i++)
{
    /* ... */
    flage = 0; /* 允许- 允许这种方式提前结束循环 */
    i = i + 3; /* 不允许 - 变更了循环次数 */
}
```

规则13.7（强制）： 不允许进行结果不会改变的布尔运算。

(13.7 Boolean operations whose results are invariant shall not be permitted.)

如果布尔运算产生的结果始终为“true”或始终为“false”，那么这很可能是编程错误。

```
enum ec { RED, BLUE, GREEN } col;
...
if (u16a < 0) /* 不允许 - 无符号16位整数总是大于等于零的 */
...
if (u16a <= 0xffff) /* 不允许-总是为真 */
...
if (s8a < 130) /* 不允许总是为真 */
...
if ( ( s8a < 10 ) && ( s8a > 20 ) ) /* 不允许-总是为假*/
...
if ( ( s8a < 10 ) || ( s8a > 5 ) ) /* 不允许-总是为真 */
...
if ( col <= GREEN ) /* 不允许-总是为真*/
...
if (s8a > 10)
{
    if (s8a > 5) /* Not compliant - s8a is not volatile */
    {
```

```
    }
}
```

14 控制流

规则 14.1（强制）： 不能有不可到达（**unreachable**）的代码。

(14.1 There shall be no unreachable code.)

本规则是针对那些在任何环境中都不能到达的代码，这些代码在编译时就能被标识出不可到达。规则排除了可以到达但永远不会执行的代码（如，保护性编程代码（**defensive programming**））。

如果从相关的入口到某部分代码之间不存在控制流路径，那么这部分代码就是不可到达的。例如，在无条件控制转移代码后的未标记代码就是不可到达的：

```
switch ( event )
{
    case E_wakeup:
        do_wakeup ();
        break; /* unconditional control transfer */
    do_more (); /* 不允许 - 不可到达代码 */
    /* ... */
    default:
        /* ... */
        break;
}
```

对整个函数来说，如果不存在调用它的手段，那么这个函数将是不可到达的。

规则14.2（强制）： 所有非空语句（**non-null statement**）应该：

- a) 不管怎样执行都至少有一个副作用（**side-effect**），或者
- b) 可以引起控制流的转移

(14.2 All non-null statements shall either:

- a) have at least one side-effect however executed, or
- b) cause control flow to change.)

任何语句（非空语句），如果既没有副作用也不引起控制流的改变，通常就会指示出编程错误，因此要进行对这样语句的静态检测。例如，下面的语句在执行时不一定带有副作用：

```
/* assume uint16_t x; and uint16_t i; */
...
x >= 3u; /* 不允许: x is compared to 3, and the answer is discarded */
```

规则 14.3（强制）： 在预处理之前，空语句只能出现在一行上；其后可以跟有注释，假设紧跟空语句的第一个字符是空格。

(14.3 Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white space character.)

通常不会故意包含空语句，但是在使用它们的地方，它们应该出现在它们本身的行上。空语句前面可以有空格以保持缩进的格式。如果一条注释跟在空语句的后面，那么至少要有

一个空格把空语句和注释分隔开来。需要这样起分隔作用的空格是因为它给读者提供了重要的视觉信息。遵循本规则使得静态检查工具能够为与其他文本出现在一行上的空语句提出警告，因为这样的情形通常表示编程错误。例如：

```
while ( ( port & 0x80 ) == 0 )
{
    ; /* wait for pin – Compliant */
    /* wait for pin */ ; /* Not compliant, comment before ; */
    ; /* wait for pin – Not compliant, no white-space char after ; */
}
```

规则14.4（强制）： 不应使用 **goto** 语句。

(14.4 The goto statement shall not be used.)

规则 14.5（强制）： 不应使用 **continue** 语句。

(14.5 The continue statement shall not be used.)

规则 14.6（强制）： 对任何迭代语句至多只应有一条**break** 语句用于循环的结束。

(14.6 For any iteration statement there shall be at most one break statement used for loop termination.)

这些规则关心的是良好的编程结构。循环中允许有一条**break** 语句，因为这允许双重结果的循环或代码优化。

规则 14.7（强制）： 一个函数在其结尾应该有单一的退出点。

(14.7 A function shall have a single point of exit at the end of the function.)

这是由 IEC 61508 良好的编程格式要求的。

规则 14.8（强制）： 组成 **switch**、**while**、**do...while** 或**for** 结构体的语句应该是复合语句。

(14.8 The statement forming the body of a switch, while, do... while or for statement shall be a compound statement.)

组成 **switch** 语句或**while**、**do ... while** 或**for** 循环结构体的语句应该是复合语句（括在大括号里），即使该复合语句只包含一条语句。

例如：

```
for ( i = 0 ; i < N_ELEMENTS ; ++i )
{
    buffer[i] = 0; /* Even a single statement must be in braces */
}
while ( new_data_available )
process_data (); /* Incorrectly not enclosed in braces */
service_watchdog (); /* Added later but, despite the appearance
                      (from the indent) it is actually not
                      part of the body of the while statement,
                      and is executed only after the loop has terminated */
```

注意，复合语句及其大括号的布局应该根据格式指南来确定。上述只是个例子。

规则 14.9（强制）： **if**（表达式）结构应该跟随有复合语句。**else** 关键字应该跟随有复合语句或者另外的**if** 语句。

(14.9 An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.)

例如：

```
if ( test1 )
{
    x = 1 ; /* Even a single statement must be in braces */
}
else if ( test2 )
{
    x = 0; /* No need for braces in else if */
}
else
    x = 3; /* This was (incorrectly) not enclosed in braces */
    y = 2; /* This line was added later but, despite the appearance
           (from the indent) it is actually not part of the else,
           and is executed unconditionally */
```

注意，复合语句及其大括号的布局应该根据格式指南来确定。上述只是个例子。

规则 14.10（强制）： 所有的 **if ... else if** 结构应该由**else** 子句结束。

(14.10 All if ... else if constructs shall be terminated with an else clause.)

不管何时一条 **if** 语句跟有一个或多个**else if** 语句都要应用本规则；最后的**else if** 必须跟有一条**else** 语句。而**if** 语句然后就是**else** 语句的简单情况不在本规则之内。

对最后的 **else** 语句的要求是保护性编程（defensive programming）。**else** 语句或者要执行适当的动作，或者要包含合适的注释以说明为何没有执行动作。这与**switch** 语句中要求具有最后一个**default** 子句（规则15.3）是一致的。

例如，下面的代码是简单的 **if** 语句：

```
if ( x > 0 )
{
    log_error (3) ;
    x = 0 ;
} /* else not needed */
```

而下面的代码描述了**if**, **else if** 结构：

```
if ( x < 0 )
{
    log_error (3);
    x = 0;
}
else if ( y < 0 )
{
    x = 3;
}
```

```

else          /* 这个else是条款所要求的, 即使 */
{             /* 程序员认为这个分支永远不会到达*/
    /* no change in value of x */
}

```

15 switch 语句

规则 15.1 (强制): **switch** 标签只能用在当最紧密封闭 (**closely-enclosing**) 的复合语句是**switch** 语句体的时候

(15.1 A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.)

case 或**default** 标签的范围应该是做为**switch** 语句体的复合语句。所有**case** 子句和**default**子句应该具有相同的范围。

规则 15.2 (强制): 每个非空的**switch** 子句应该无条件的被**break** 语句终止。

(15.2 An unconditional break statement shall terminate every non empty switch clause.)

每个**switch** 子句中的最后一条语句应该是**break** 语句, 或者如果**switch** 子句是复合语句, 那么复合语句的最后一条语句应该是**break** 语句。

规则 15.3 (强制): **switch** 语句的最后子句应该是**default** 子句。

(15.3 The final clause of a switch statement shall be the default clause.)

对最后的 **default** 子句的要求是出于保护性编程。该子句应该执行适当的动作, 或者包含合适的注释以说明为何没有执行动作。

规则 15.4 (强制): **switch** 表达式不应是有效的布尔值。

(15.4 A switch expression shall not represent a value that is effectively Boolean.)

例如:

```

switch ( x == 0) /* 不允许 - effectively Boolean */
{
    ...
}

```

规则15.5 (强制): 每个 **switch** 语句至少应有一个**case** 子句。

(15.5 Every switch statement shall have at least one case clause.)

例如:

```

switch (x)
{
    uint8_t var; /* 不允许 */
    case 0:
        a = b;
        break; /* break is required here */
    case 1: /* 空句子, 不需要break*/
    case 2:
        a = c; /* executed if x is 1 or 2 */
}

```

```

        if ( a == b )
        {
case 3: /* 不允许 – case 不允许放在这儿 (if语句中) */
        }
        break;
case 4:
        a = b; /* not compliant – non empty drop through */
case 5:
        a = c;
        break;
default: /* default clause is required */
        errorflag = 1; /* should be non-empty if possible */
        break; /* break is required here, in case a
                future modification turns this into a
                case clause */
    }

```

16 函数

规则 16.1（强制）： 函数定义不得带有可变数量的参数

(16.1 Functions shall not be defined with a variable number of arguments.)

本特性存在许多潜在的问题。用户不应编写使用可变数量参数的附加函数。这排除了 `stdarg.h`、`va_arg`、`va_start` 和 `va_end` 的使用。一个使用可变参数的例子是 `printf` 函数。

规则 16.2（强制）： 函数不能调用自身，不管是直接还是间接的。

(16.2 Functions shall not call themselves, either directly or indirectly.)

这意味着在安全相关的系统中不能使用递归函数调用。递归本身承载着可用堆栈空间过度的危险，这能导致严重的错误。除非递归经过了非常严格的控制，否则不可能在执行之前确定什么是最坏情况（**worst-case**）的堆栈使用。

规则 16.3（强制）： 在函数的原型声明中应该为所有参数给出标识符

(16.3 Identifiers shall be given for all of the parameters in a function prototype declaration.)

出于兼容性、清晰性和可维护性的原因，应该在函数的原型声明中为所有参数给出名字。

规则 16.4（强制）： 函数的声明和定义中使用的标识符应该一致

(16.4 The identifiers used in the declaration and definition of a function shall be identical.)

规则 16.5（强制）： 不带参数的函数应当声明为具有 **void** 类型的参数

(16.5 Functions with no parameters shall be declared with parameter type void.)

函数应该声明为具有返回类型（见规则8.2），如果函数不返回任何数据，返回类型为 **void**。类似地，如果函数不带参数，参数列表应声明为 **void**。例如函数 `myfunc`，如果既不带参数也不返回数据则应声明为：

void myfunc (void);

规则16.6（强制）： 传递给一个函数的参数应该与声明的参数匹配。

(16.6 The number of arguments passed to a function shall match the number of parameters.)

这个问题可以通过使用函数原型完全避免——见规则8.1。本规则被保留是因为编译器可能不会标记这样的约束错误。

规则 16.7（建议）： 函数原型中的指针参数如果不是用于修改所指向的对象，就应该声明为指向**const** 的指针。

(16.7 A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.)

本规则会产生更精确的函数接口定义。**const** 限定应当用在所指向的对象而非指针，因为要保护的是对象本身。

例如：

```
void myfunc ( int16_t *param1, const int16_t *param2, int16_t *param3 )
/* 参数一：对象的地址是可以修改的 – 不是一个常量
   参数二：Addresses an object which is not modified – const required
   param3：Addresses an object which is not modified – 缺少const */
{
    *param1 = *param2 + *param3;
    return;
}
/* 参数三应该加上const */
```

规则16.8（强制）： 带有 **non-void** 返回类型的函数其所有退出路径都应具有显式的带表达式的**return** 语句。

(16.8 All exit paths from a function with non-void return type shall have an explicit return statement with an expression.)

表达式给出了函数的返回值。如果**return** 语句不带表达式，将导致未定义的行为（而且编译器不会给出错误）。

规则 16.9（强制）： 函数标识符的使用只能或者加前缀**&**，或者使用括起来的参数列表，列表可以为空。

(16.9 A function identifier shall only be used with either a preceding **&**, or with a parenthesized parameter list, which may be empty.)

如果程序员写为：

```
if (f) /* 不允许–g应使用 f() 或者 &f */
{
    /* ... */
}
```

那么就不会清楚其意图是要测试函数的地址是否为NULL，还是执行函数f()的调用。

规则 16.10（强制）： 如果函数返回了错误信息，那么错误信息应该进行测试。

(16.10 If a function returns error information, then that error information shall be tested.)

e tested.)

一个函数（标准库中的函数、第三方库函数、或者是用户定义的函数）能够提供一些指示错误发生的方法。这可以通过使用错误标记、特殊的返回数据或者其他手段。不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

然而要注意到，相对于在函数完成后才检测错误的做法（见规则20.3）来说，**对函数输入值的检查是更为鲁棒的防止错误的手段**。还要注意到，对`errno` 的使用（为了返回函数的错误信息）是笨拙的并应该谨慎使用（见规则20.5）。

17 指针和数组

规则 17.1（强制）： 指针的数学运算只能用在指向数组或数组元素的指针上。

(17.1 Pointer arithmetic shall only be applied to pointers that address an array or array element.)

对并非指向数组或数组元素的指针做整数加减运算（包括增值和减值）会导致未定义的行为。

规则 17.2（强制）： 指针减法只能用在指向同一数组中元素的指针上。

(17.2 Pointer subtraction shall only be applied to pointers that address elements of the same array.)

只有当两个指针指向（或至少好象是指向了）同一数组内的对象时，指针减法才能给出良好定义的结果。

规则 17.3（强制）： `>`、`>=`、`<`、`<=` 不应用在指针类型上，除非指针指向同一数组。

(17.3 `>`, `>=`, `<`, `<=` shall not be applied to pointer types except where they point to the same array.)

如果两个指针没有指向同一个对象，那么试图对指针做比较将导致未定义的行为。注意：允许指向超出数组尾部的元素，但对该元素的访问是禁止的。

规则 17.4（强制）： 数组的索引应当是指针数学运算的唯一可允许的方式

(17.4 Array indexing shall be the only allowed form of pointer arithmetic.)

数组的索引是指针数学运算的唯一可接受的方式，因为它比其他的指针操作更为清晰并由此具有更少的错误倾向。本规则禁止了指针数值的显式运算。数组索引只能应用在定义为数组类型的对象上。任何显式计算的指针值潜在地会访问不希望访问的或无效的内存地址。指针可以超出数组或结构的范围，或者甚至可以有效地指向任意位置。见规则21.1。

```
void my_fn(uint8_t * p1, uint8_t p2[])
{
    uint8_t index = 0;
    uint8_t *p3;
    uint8_t *p4;
    *p1 = 0;
    p1++; /* 不允许 - pointer increment */
    p1 = p1 + 5; /* 不允许- pointer increment */
    p1[5] = 0; /* 不允许 - p1并不是明确的数组 */
    p3 = &p1[5]; /*不允许 - p1并不是明确的数组*/
```

```

    p2[0] = 0;
    index ++;
    index = index + 5;
    p2[index] = 0; /* 允许*/
    p4 = &p2[5]; /* 允许 */
}
uint8_t a1[16];
uint8_t a2[16];
my_fn (a1, a2);
my_fn (&a1[4], &a2[4]);
uint8_t a[10];
uint8_t *p;
p = a;
* (p + 5) = 0; /* 不允许 */
p[5] = 0; /* 允许*/

```

规则17.5（建议）： 对象声明所包含的间接指针不得多于2 级

（17.5 The declaration of objects should contain no more than 2 levels of pointer indirection.）

多于 2 级的间接指针会严重削弱对代码行为的理解，因此应该避免。

```

typedef int8_t * INTPTR;
struct s {
    int8_t * s1; /* 允许 */
    int8_t * s2; /* compliant */
    int8_t * s3; /* compliant */
};
struct s * ps1; /* compliant */
struct s ** ps2; /* compliant */
struct s *** ps3; /* 不允许 */
int8_t ** ( *pfunc1 ) (); /* compliant */
int8_t ** ( **pfunc2 ) (); /* compliant */
int8_t ** ( ***pfunc3 ) (); /* not compliant */
int8_t *** ( ** pfunc4 ) (); /* not compliant */
void function ( int8_t * par1,
               int8_t ** par2,
               int8_t *** par3, /* not compliant */
               INTPTR * par4,
               INTPTR * const * const par5, /* not compliant */
               int8_t * par6[],
               int8_t ** par7[] ) /* not compliant */
{
    int8_t * ptr1 ;
    int8_t ** ptr2 ;
    int8_t *** ptr3 ; /* not compliant */

```

```

    INTPTR * ptr4;
    INTPTR * const * const ptr5; /* not compliant */
    int8_t * ptr6[10];
    int8_t ** ptr7[10];
}

```

类型解释：

- par1 和ptr1 是指向int8_t 的指针。
- par2 和ptr2 是指向int8_t 的指针的指针。
- par3 和ptr3 是指向int8_t 的指针的指针的指针。这是三级，因此不适合。
- par4 和ptr4 扩展开是指向int8_t 的指针的指针。
- par5 和ptr5 扩展开是指向int8_t 的指针的const 指针的const 指针。这是三级，因此不合适。
- par6 是指向int8_t 的指针的指针，因为数组被转化成指向数组初始元素的指针。
- ptr6 是int8_t 类型的指针数组。
- par7 是指向int8_t 的指针的指针的指针，因为数组被转化成指向数组初始元素的指针。这是三级，因此不合适。
- ptr7 是指向int8_t 的指针的指针数组。这是合适的。

规则 17.6（强制）： 自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象。

（17.6 The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.）

如果一个自动对象的地址赋值给其他的或者具有更大作用域的对象、或者静态对象、或者从一个函数返回的对象，那么在初始对象停止存在（其地址成为无效）时，包含地址的对象可能会延续存在。

例如：

```

int8_t * foobar (void)
{
    int8_t local_auto;
    return (&local_auto); /* 不允许 */
}

```

18 结构与联合

规则 18.1（强制）： 所有结构与联合类型应该在转换单元（translation unit）的结尾是完善的。

（18.1 All structure and union types shall be complete at the end of a translation unit.）

结构或联合的完整声明应该包含在任何涉及该结构的转换单元之内。

```

struct tnode * pt; /* 声明，此刻tnode是不完善的*/
struct tnode
{
    int count;
    struct tnode * left;
    struct tnode * right;
}

```

```
}; /*tnode到现在是完善的 */
```

规则18.2（强制）： 对象不能赋值给重叠（overlapping）对象。

（18.2 An object shall not be assigned to an overlapping object.）

当两个对象创建后，如果它们拥有重叠的内存空间并把一个拷贝给另外一个时，该行为是未定义的。

规则 18.3（强制）： 不能为了不相关的目的重用一块内存区域。

（18.3 An area of memory shall not be reused for unrelated purposes.）

本规则涉及了使用内存存储某些数据并在程序仍然运行期间的其他时刻使用同一块内存存储不相关数据的技术。很明显这依赖于两段不同的数据块，它们存在于程序运行期间不相连的时间段而且从来不会被同时请求。

对于安全相关的系统，不建议这样做，因为它会带来许多危险。比如：一个程序可能试图访问某个区域的某种类型的数据，而当时该区域正在存储其他类型的数据（如，由中断引起的）。这两种类型的数据在存储上可能不同而且可能会侵占其他数据。这样在每次切换使用时，数据可能不会正确初始化。这样的做法在并发系统中尤其是危险的。

然而要了解有时出于效率的原因可能会要求这样的存储共享。此时就非常需要采取检测手段以确保错误类型的数据永远不会被访问，确保数据始终被适当地初始化了以及确保对其他部分数据（如，由分配不同引起的）的访问是不可能的。所采取的检测手段应该归纳为文档，并在违背本规则的背离内被证明是正确的。

可以使用联合或其他手段做到这一点。

注意，MISRA-C 不想对编译器翻译源代码的实际行为给出限制，因为用户通常对此没有有效的控制。所以，举例来说，编译器中的内存分配，不管是动态堆、动态堆栈或静态的，可能会仅仅因为轻微的代码变化就发生显著的改变，即使是在相同的优化级别上。还要注意，某些优化会合理地产生，甚至在用户没有做出这样的请求时。

规则 18.4（强制）： 不要使用联合。

（18.4 Unions shall not be used.）

规则18.3 禁止为不相关的目的重用内存区域，然而，即使内存区域是为了相关目的重用的，仍然会存在数据被误解的风险。因此，提出本规则禁止针对任何目的而使用联合。尽管如此，要认识到在某些情况下需要谨慎地使用联合以构建有效率的实现。发生这些情况时，如果所有相关的定义实现的行为被归纳为文档了，那么对本规则的违背也是可以接受的。实践中，可以在设计文档内指出编译器手册的实现章节。与此有关的实现行为的种类是：

- 填充——在联合的结尾插入了多少填充
- 排列——在联合中排列了多少结构成员
- 存储次序（endianess）——数据字中最有效的字节是存储在最低内存地址还是最高地址
- 位次序（bit-order）——字节中的位是如何计数的以及如何分配在位域中的

对如下情况违背规则是可以接受的：（a）数据的打包和解包，如在发送和接收消息时；（b）实现变量录取（variant records），假设变量是由通用的域（common fields）区分的。不带区分器（differentiator）的变量录取在任何情况下都是不合适的。

数据打包和解包

在本例中，使用联合访问一个32 位数据字的字节以存储从高字节优先的网络上接收到

的字节。这种特殊假设：

- uint32_t 类型占据32 位
- uint8_t 类型占据8 位
- 实现把数据字的最高有效字节存储在最低的内存地址

实现接收和打包的代码如下：

```
typedef union {
    uint32_t word;
    uint8_t bytes[4];
} word_msg_t;

uint32_t read_word_big_endian (void)
{
    word_msg_t tmp;
    tmp.bytes[0] = read_byte ();
    tmp.bytes[1] = read_byte ();
    tmp.bytes[2] = read_byte ();
    tmp.bytes[3] = read_byte ();
    return (tmp.word);
}
```

值得注意的是上面的函数体可以写成如下可移植的形式：

```
uint32_t read_word_big_endian (void)
{
    uint32_t word;
    word = ( ( uint32_t ) read_byte () ) << 24;
    word = word | ( ( ( uint32_t ) read_byte () ) << 16 );
    word = word | ( ( ( uint32_t ) read_byte () ) << 8 );
    word = word | ( ( uint32_t ) read_byte () );
    return (word);
}
```

不幸的是，面临可移植的实现时，大多数编译器产生的代码远非有效率的。当程序对高执行速度和低内存使用的要求胜于移植性时，使用联合的实现是可以考虑的。

变量录取

联合通常用于实现变量录取。每个变量享有共同的域并具有其特有的附加域。本例基于CAN 总线校准协议(CAN Calibration Protocol, CCP)，其中每个发送给CCP 客户端的CAN 消息共享两个通用的域，每个域占一个字节。其后最多可附加6 个字节，这些字节的解释依赖于存储在第一个字节中的消息类型。

这个特定实现依赖于如下假设：

- uint16_t 类型占据16 位
- uint8_t 类型占据8 位
- 排列和打包的规则是，uint8_t 和uint16_t 类型的结构成员之间不存在间隙

为了简化，本例中只考虑两种消息类型。这里给出的代码是不完整的，只是用来描述变量录取而不是做为CCP 的实现模块。

```
/* The fields common to all CCP messages */
typedef struct {
    uint8_t msg_type;
```

```
        uint8_t sequence_no;
    } ccp_common_t;
    /* CCP connect message */
    typedef struct {
        ccp_common_t common_part;
        uint16_t station_to_connect;
    } ccp_connect_t;
    /* CCP disconnect message */
    typedef struct {
        ccp_common_t common_part;
        uint8_t disconnect_command;
        uint8_t pad;
        uint16_t station_to_disconnect;
    } ccp_disconnect_t;
    /* The variant */
    typedef union {
        ccp_common_t common;
        ccp_connect_t connect;
        ccp_disconnect_t disconnect;
    } ccp_message_t;
    void process_ccp_message (ccp_message_t *msg)
    {
        switch (msg->common.msg_type)
        {
            case Ccp_connect:
                if (MY_STATION == msg->connect.station_to_connect)
                {
                    ccp_connect ();
                }
                break;
            case Ccp_disconnect:
                if (MY_STATION == msg->disconnect.station_to_disconnect)
                {
                    if (PERM_DISCONNECT ==
msg->disconnect.disconnect_command)
                    {
                        ccp_disconnect ();
                    }
                }
                break;
            default:
                break; /* ignore unknown commands */
        }
    }
```

19 预处理指令

规则 19.1（建议）： 文件中的**#include** 语句之前只能是其他预处理指令或注释。

(19.1 #include statements in a file should only be preceded by other preprocessor directives or comments.)

代码文件中所有**#include** 指令应该成组放置在接近文件顶部的位置。本规则说明，文件中可以优先**#include** 语句放置的只能是其他预处理指令或注释。

规则 19.2（建议）： **#include** 指令中的头文件名字里不能出现非标准字符。

(19.2 Non-standard characters should not occur in header file names in #include directives.)

如果在头文件名字预处理标记的 < 和 > 限定符或 " 和 " 限定符之间使用了 ' , \ , 或 /* 字符，该行为是未定义的。

规则 19.3（强制）： **#include** 预处理指令应该跟随<filename>或"filename"序列。

(19.3 The #include directive shall be followed by either a <filename> or "filename" sequence.)

例如，如下语句是允许的：

```
#include "filename.h"
#include <filename.h>
#define FILE "filename.h"
#include FILE
```

规则19.4（强制）： C的宏只能扩展为用大括号括起来的初始化、常量、小括号括起来的表达式、类型限定符、存储类标识符或do-while-zero 结构。

(19.4 C macros shall only expand to a braced initialiser, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.)

这些是宏当中所有可允许使用的形式。存储类标识符和类型限定符包括诸如extern、static和const 这样的关键字。使用任何其他形式的#define 都可能导致非预期的行为，或者是非常难懂的代码。

特别的，宏不能用于定义语句或部分语句，除了 do-while 结构。宏也不能重定义语言的语法。宏的替换列表中的所有括号，不管哪种形式的 ()、{}、[] 都应该成对出现。

do-while-zero 结构（见下面的例子）是在宏语句体中唯一可接受的具有完整语句的形式。

do-while-zero 结构用于封装语句序列并确保其是正确的。注意：在宏语句体的末尾必须省略分号。

例如：

```
/* 一下语句是合适的 */
#define PI 3.14159F          /* 常量 */
#define XSTAL 1000000        /* Constant */
#define CLOCK (XSTAL / 16)  /*常数表达式 */
#define PLUS2(X) ((X) + 2)   /* 宏扩展表达式 */
#define STOR extern          /* 存储类型说明符 */
#define INIT(value) { (value), 0, 0 } /* 初始化 */
#define READ_TIME_32 () \
```

```

do { \
    DISABLE_INTERRUPTS (); \
    time_now = (uint32_t) TIMER_HI << 16; \
    time_now = time_now | (uint32_t) TIMER_LO; \
    ENABLE_INTERRUPTS (); \
} while (0)          /* do-while-zero的例子 */
/* 以下语句不允许 */
#define int32_t long /* 应使用 typedef 代替 */
#define STARTIF if (      /* 不稳定的 () 并且重新定义了语言 */

```

规则19.5（强制）： 宏不能在块中进行 `#define` 和 `#undef`。

(19.5 Macros shall not be `#define`'d or `#undef`'d within a block.)

C 语言中，尽管在代码文件中的任何位置放置`#define` 或`#undef` 是合法的，但把它们放在块中会使人误解为好像它们存在于块作用域。

通常，`#define` 指令要放在接近文件开始的地方，在第一个函数定义之前。而`#undef` 指令通常不一定需要（见规则19.6）。

规则 19.6（强制）： 不要使用`#undef`。

(19.6 `#undef` shall not be used.)

通常，`#undef` 是不需要的。当它出现在代码中时，能使宏的存在或含义产生混乱。

规则 19.7（建议）： 函数的使用优先选择函数宏（**function-like macro**）。

(19.7 A function should be used in preference to a function-like macro.)

)

由于宏能提供比函数优越的速度，函数提供了一种更为安全和鲁棒的机制。在进行参数的类型检查时尤其如此。函数宏的问题在于它可能会多次计算参数。

规则 19.8（强制）： 函数宏的调用不能缺少参数

(19.8 A function-like macro shall not be invoked without all of its arguments.)

这是一个约束错误，但是预处理器知道并忽略此问题。函数宏中的每个参数的组成必须至少有一个预处理标记，否则其行为是未定义的。

规则 19.9（强制）： 传递给函数宏的参数不能包含看似预处理指令的标记。

(19.9 Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.)

如果任何参数的行为类似预处理指令，使用宏替代函数时的行为将是不可预期的。

规则 19.10（强制）： 在定义函数宏时，每个参数实例都应该以小括号括起来，除非它们做为`#`或`##`的操作数。

(19.10 In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`.)

函数宏的定义中，参数应该用小括号括起来。例如一个`abs` 函数可以定义成：

```
#define abs(x) ((x) >= 0) ? (x) : -(x)
```

不能定义成：

```
#define abs(x) ((x) >= 0) ? x : -x)
```

如果不坚持本规则，那么当预处理器替代宏进入代码时，操作符优先顺序将不会给出要求的结果。

考虑前面第二个不正确的定义被替代时会发生什么：

```
z = abs(a - b);
```

将给出如下结果：

```
z = ((a - b >= 0) ? a - b : -a - b);
```

子表达式 $-a - b$ 相当于 $(-a) - b$ ，而不是希望的 $-(a - b)$ 。把所有参数都括进小括号中就可以避免这样的问题。

规则 19.11(强制)： 预处理指令中所有宏标识符在使用前都应先定义，除了**#ifdef** 和**#ifndef** 指令及**defined()**操作符。

(19.11 All macro identifiers in preprocessor directives shall be defined before use, except in **#ifdef** and **#ifndef** preprocessor directives and the **defined()** operator)

如果试图在预处理指令中使用未经定义的标识符，预处理器有时不会给出任何警告但会假定其值为零。**#ifdef**、**#ifndef** 和**defined()**用来测试宏是否存在并由此进行排除。

例如：

```
#if x < 0 /* x assumed to be zero if not defined */
```

在标识符被使用之前要考虑使用**#ifdef** 进行测试。

注意，预处理标识符可以使用**#define** 指令来定义也可以在编译器调用所指定的选项中定义。然而更多的是使用**#define** 指令。

规则 19.12（强制）： 在单一的宏定义中最多可以出现一次 **#** 或 **##** 预处理器操作符。

(19.12 There shall be at most one occurrence of the **#** or **##** operators in a single macro definition.

)

与 **#** 或 **##** 预处理器操作符相关的计算次序如果未被指定则会产生问题。为避免该问题，在单一的宏定义中只能使用其中一种操作符（即，一个 **#**、或一个 **##**、或都不用）。

规则 19.13（建议）： 不要使用**#** 或 **##** 预处理器操作符。

(19.13 The **#** and **##** operators should not be used.)

与 **#** 或 **##** 预处理器操作符相关的计算次序如果未被指定则会产生问题。编译器对这些操作符的实现是不一致的。为避免这些问题，最好不要使用它们。

规则 19.14（强制）： **defined** 预处理操作符只能使用两种标准形式之一。

(19.14 The **defined** preprocessor operator shall only be used in one of the two standard forms.)

defined 预处理操作符的两种可允许的形式为：

```
defined(identifier)
```

```
defined identifier
```

任何其他的形式都会导致未定义的行为，比如：

```
#if defined(X > Y) /* not compliant – undefined behaviour */
```

在 **#if** 或 **#elif** 预处理指令的扩展中定义的标记也会导致未定义的行为，应该避免。如：

```
#define DEFINED defined
```

```
#if DEFINED(X) /* not compliant – undefined behaviour */
```

规则19.15（强制）： 应该采取防范措施以避免一个头文件的内容被包含两次。

(19.15 Precautions shall be taken in order to prevent the contents of a header file being included twice.)

当转换单元（translation unit）包含了复杂层次的嵌套头文件时，会发生某头文件被包含多于一次的情形。这最多会导致混乱。如果它导致了多个定义或定义冲突，其结果将是未定义的或者是错误的行为。

多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取阻止头文件内容被包含多于一次的机制。通常的手段是为每个文件配置一个宏；当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。

例如，一个名为“ahdr.h”的文件可以组织如下：

```
#ifndef AHDR_H
#define AHDR_H
/* The following lines will be excluded by the preprocessor if the file is included
more than once */
...
#endif
或者可以使用下面的形式
#ifdef AHDR_H
#error Header file is already included
#else
#define AHDR_H
/* The following lines will be excluded by the preprocessor if the file is included more
than once */
...
#endif
```

规则19.16（强制）： 预处理指令在句法上应该是有意义的，即使是在被预处理器排除的情况下。

(19.16 Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.)

当一段源代码被预处理指令排除时，每个被排除语句的内容被忽略直到出现一个 #else、#elif 或 #endif 指令（根据上下文内容）。如果其中一个被排除指令的组成形式不好（badly formed），编译器忽略它时不会给出任何警告，这将带来不幸的后果。

本规则要求所有预处理指令在句法上是有效的，即使它们出现在被排除的代码块中。特别地，要确保 #else 和 #endif 指令后不要跟随除空格之外的任何字符。在强制执行这个ISO 要求时编译器并非始终一致。

```
#define AAA 2
...
int foo (void)
{
    int x = 0 ;
    ...
    #ifndef AAA
    x = 1 ;
    #else1 /*不允许*/
```

```

        x = AAA ;
    #endif
    ...
    return x ;
}

```

规则**19.17**（强制）：所有的 **#else**、**#elif** 和 **#endif** 预处理指令应该同与它们相关的 **#if** 或 **#ifdef** 指令放在相同的文件中。

（19.17 All **#else**, **#elif** and **#endif** preprocessor directives shall reside in the same file as the **#if** or **#ifdef** directive to which they are related.）

当语句块的包含和排除是被一系列预处理指令控制时，如果所有相关联的指令没有出现在同一个文件中就会产生混乱。本规则要求所有的预处理指令序列 **#if / #ifdef ... #elif ... #else ... #endif** 应该放在同一个文件中。遵循本规则会保持良好的代码结构并能避免维护性问题。

注意，这并不排除把所有这样的指令放在众多被包含文件中的可能性，只要与某一序列相关的所有指令放在一个文件中即可。

```

file.c
#define A
...
#ifdef A
...
#include "file1.h"
#endif
...
#if 1
#include "file2.h"
...
EOF
file1.h
#if 1
...
#endif /* 允许 */
EOF
file2.h
...
#endif /*不允许*/

```

6.20 标准库

规则 **20.1**（强制）：标准库中保留的标识符、宏和函数不能被定义、重定义或取消定义。

（20.1 Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.）

通常 **#undef** 一个定义在标准库中的宏是件坏事。同样不好的是，**#define** 一个宏名字，而该名字是C 的保留标识符或者标准库中做为宏、对象或函数名字的C 关键字。例如，存

在一些特殊的保留字和函数名字，它们的作用为人所熟知，如果对它们重新定义或取消定义就会产生一些未定义的行为。这些名字包括`defined`、`__LINE__`、`__FILE__`、`__DATE__`、`__TIME__`、`__STDC__`、`errno` 和`assert`。

`#undef` 的使用也可以参见规则19.6。

ISO C 的保留标识符在本文档中参见7.1.3 节和7.13 节关于ISO 9899 :1990 [2]，同时也应该被编译器的编写者写入文档。通常，所有以下划线开始的标识符都是保留的。

规则 20.2（强制）： 不能重用标准库中宏、对象和函数的名字。

(20.2 The names of standard library macros, objects and functions shall not be reused.)

如果程序员使用了标准库中宏、对象或函数的新版本（如，功能增强或输入值检查），那么更改过的宏、对象或函数应该具有新的名字。这是用来避免不知是使用了标准的宏、对象或函数还是使用了它们的更新版本所带来的任何混淆。所以，举例来说，如果`sqrt` 函数的新版本被写做检查输入值非负，那么这新版本不能命名为“`sqrt`”而应该给出新的名字。

规则 20.3（强制）： 传递给库函数的值必须检查其有效性。

(20.3 The validity of values passed to library functions shall be checked.)

C 标准库中的许多函数根据ISO 标准 [2] 并不需要检查传递给它们的参数的有效性。即使标准要求这样，或者编译器的编写者声明要这么做，也不能保证会做出充分的检查。因此，程序员应该为所有带有严格输入域的库函数（标准库、第三方库及自己定义的库）提供适当的输入值检查机制。

具有严格输入域并需要检查的函数例子为：

- `math.h` 中的许多数学函数，比如：

- 负数不能传递给 `sqrt` 或`log` 函数；

- `fmod` 函数的第二个参数不能为零

- `toupper` 和`tolower`：当传递给`toupper` 函数的参数不是小写字母时，某些实现能产生并非预期的结果（`tolower` 函数情况类似）

- 如果为 `ctype.h` 中的字符测试函数传递无效的值时会给出未定义的行为

- 应用于大多数负整数的 `abs` 函数给出未定义的行为

在 `math.h` 中，尽管大多数数学库函数定义了它们允许的输入域，但在域发生错误时它们的返回值仍可能随编译器的不同而不同。因此，对这些函数来说，预先检查其输入值的有效性就变得至关重要。

程序员在使用函数时，应该识别应用于这些函数之上的任何的域限制（这些限制可能会也可能不会在文档中说明），并且要提供适当的检查以确认这些输入值位于各自域中。当然，在需要时，这些值还可以更进一步加以限制。

有许多方法可以满足本规则的要求，包括：

- 调用函数前检查输入值

- 设计深入函数内部的检查手段。这种方法尤其适应于实验室内开发的库，纵然它也可以用于买进的第三方库（如果第三方库的供应商声明他们已内置了检查的话）。

- 产生函数的“封装”（`wrapped`）版本，在该版本中首先检查输入，然后调用原始的函数。

- 静态地声明输入参数永远不会采取无效的值。

注意，在检查函数的浮点参数时（浮点参数在零点上为奇点），适当的做法是执行其是否为零的检查。这对规则13.3 而言是可以接受的例外，不需给出背离。然而如果当参数趋

近于零时，函数值的量级趋近无穷的话，仍然有必要检查其在零点（或其他任何奇点）上的容限，这样可以避免溢出的发生。

规则 20.4（强制）： 不能使用动态堆的内存分配

（20.4 Dynamic heap memory allocation shall not be used.）

这排除了对函数`alloc`、`malloc`、`realloc` 和`free` 的使用。

涉及动态内存分配时，存在整个范围内的未指定的、未定义的和实现定义的行为，以及其他大量的潜在缺陷。动态堆内存分配能够导致内存泄漏、数据不一致、内存耗尽和不确定的行为。

注意，某些实现可能会使用动态堆内存的分配以实现其他函数（如库`string.h` 中的函数）。如果这种情况发生，也需要避免使用这些函数。

规则 20.5（强制）： 不要使用错误指示`errno`。

（20.5 The error indicator `errno` shall not be used.）

`errno` 做为C 的简捷工具，在理论上是有用的，但在实际中标准没有很好地定义它。一个非零值可以指示问题的发生，也可以不用它指示；做为结果不应该使用它。即使对于那些已经良好定义了`errno` 的函数而言，宁可在调用函数前检查输入值也不依靠`errno` 来捕获错误（见规则16.10）。

规则 20.6（强制）： 不应使用库`<stddef.h>`中的宏`offsetof`。

（20.6 The macro `offsetof`, in library `<stddef.h>`, shall not be used.）

当这个宏的操作数的类型不兼容或使用了位域时，它的使用会导致未定义的行为。

规则 20.7（强制）： 不应使用 `setjmp` 宏和`longjmp` 函数

（20.7 The `setjmp` macro and the `longjmp` function shall not be used.）

`setjmp` 和`longjmp` 允许绕过正常的函数调用机制，不应该使用。

规则 20.8（强制）： 不应使用信号处理工具`<signal.h>`

（20.8 The signal handling facilities of `<signal.h>` shall not be used. ）

信号处理包含了实现定义的和未定义的行为。

规则 20.9（强制）： 在产品代码中不应使用输入/输出库`<stdio.h>`。

（20.9 The input/output library `<stdio.h>` shall not be used in production code. ）

这包含文件和I/O 函数`fgetpos`、`fopen`、`ftell`、`gets`、`perror`、`remove`、`rename` 和`ungetc`。流和文件I/O 具有大量未指定的、未定义的和实现定义的行为。本文档中假定正常情况下嵌入式系统的产品代码中不需要它们。

如果产品代码中需要 `stdio.h` 中的任意特性，那么需要了解与此特性相关的某些问题。

规则 20.10（强制）： 不应使用库`<stdlib.h>`中的函数`atof`、`atoi` 和`atol`。

（20.10 The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.）

当字符串不能被转换时，这些函数具有未定义的行为。

规则 20.11（强制）： 不应使用库`<stdlib.h>`中的函数`abort`、`exit`、`getenv` 和`system`。

(20.11 The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.)

正常情况下，嵌入式系统不需要这些函数，因为嵌入式系统一般不需要同环境进行通讯。如果一个应用中必需这些函数，那么一定要在所处环境中检查这些函数的实现定义的行为。

规则 20.12 (强制)： 不应使用库`<time.h>`中的时间处理函数。

(20.12 The time handling functions of library `<time.h>` shall not be used.)

包括`time`、`strftime`。这个库同时钟有关。许多方面都是实现定义的或未指定的，如时间的格式。如果要使用`time.h` 中的任一功能，那么必须要确定所用编译器对它的准确实现，并给出背离。

21. 运行时错误

规则 21.1 (强制)： 最大限度降低运行时错误必须要确保至少使用了下列方法之一：

- a) 静态分析工具/技术；
- b) 动态分析工具/技术；
- c) 显式的代码检测以处理运行时故障

(21.1 Minimisation of run-time failures shall be ensured by the use of at least one of

- a) static analysis tools/techniques;
- b) dynamic analysis tools/techniques;
- c) explicit coding of checks to handle run-time faults.)

运行时检测是个问题，它不专属于C，但C 程序员需要加以特别的注意。这是因为C 语言在提供任何运行时检测方面能力较弱。对于鲁棒的软件来说，动态检测是必需的，但C 的实现不需要。因此C 程序员需要谨慎考虑的问题是，在任何可能出现运行时错误的地方增加代码的动态检测。

当表达式仅仅由处在良好定义范围内的值组成时，倘若可以声称，对于所有处在定义范围内的值来说不会发生异常的话，运行时检测就不是必需的。如果使用了这样的声明，它应该与它所依赖的假设一起组织成文档。然而如果使用了这种方法，一定要小心的是，后续的代码改变可能会使原先的假设无效，或者要小心出于某种原因而改变了原先的假设。

下面的条款给出了在何处需要考虑提供动态检测的指导：

□ 数学运算错误

它包括表达式运算错误，如溢出、下溢出、零除或移位时有效位的丢失。

考虑整数溢出，注意，无符号的整数计算不会产生严格意义上的溢出（产生未定义的值），但是会产生值的折叠（`wrap-around`）（产生经过定义的但可能是错误的值）。

□ 指针运算

确保在动态计算一个地址时，被计算的地址是合理的并指向某个有意义的地方。特别要保证指向一个结构或数组的内部，那么当指针增加或者改变后仍然指向同一个结构或数组。参见指针运算的限制——规则17.1、17.2、17.4。

□ 数组界限错误

在使用数组索引元素前要确保它处于数组大小的界限之内

□ 函数参数

见规则 20.3

□ 引用指针的值（**pointer dereferencing**）

如果一个函数返回一个指针，而接下来该指针的值被引用，那么程序首先要检查指

针不是NULL。在一个函数内部，指出哪个指针可以保持NULL 哪个指针不可以保持NULL 是相对简单的事情。而跨越函数界限，尤其是在调用定义在其他文件或库中的函数时，这就困难得多。

```
/* Given a pointer to a message, check the message header and return  
* a pointer to the body of the message or NULL if the message is  
* invalid */  
const char_t *msg_body (const char_t *msg)  
{  
    const char_t *body = NULL;  
    if (msg != NULL)  
    {  
        if (msg_header_valid (msg) )  
        {  
            body = &msg[MSG_HEADER_SIZE];  
        }  
    }  
    return (body);  
}  
...  
char_t msg_buffer[MAX_MSG_SIZE];  
const char_t *payload;  
...  
payload = msg_body (msg_buffer);  
if (payload != NULL)  
{  
    /* process the message payload */  
}
```

用于最小化运行时错误的技术应该详细计划并写成文档，比如，在设计标准、测试计划、静态分析配置文件、代码检查清单中。