1. 简介

市面上介绍 C 语言以及编程方法的书数目繁多，但对如何编写优质嵌入式 C 程序却鲜有介绍，特别是对应用于单片机、ARM7、Cortex-M3 这类微控制器上的优质 C 程序编写方法几乎是个空白。本文面向的，正是使用单片机、ARM7、Cortex-M3 这类微控制器的底层编程人员。

编写优质嵌入式 C 程序绝非易事，它跟设计者的思维和经验积累关系密切。嵌入式 C 程序员不仅需要熟知硬件的特性、硬件的缺陷等，更要深入一门语言编程，不浮于表面。为了更方便的操作硬件，还需要对编译器进行深入的了解。

本文将从语言特性、编译器、防御性编程、测试和编程思想这几个方面来讨论如何编写优质嵌入式 C 程序。与很多杂志、书籍不同，本文提供大量真实实例、代码段和参考书目，不仅介绍应该做什么，还重点介绍如何做、以及为什么这样做。编写优质嵌入式 C 程序涉及面十分广，需要程序员长时间的经验积累，本文希望能缩短这一过程。

2. C 语言特性

语言是编程的基石，C 语言诡异且有种种陷阱和缺陷，需要程序员多年历练才能达到较为完善的地步。虽然有众多书籍、杂志、专题讨论过 C 语言的陷阱和缺陷，但这并不影响本节再次讨论它。总是有大批的初学者，前仆后继的倒在这些陷阱和缺陷上，民用设备、工业设备甚至是航天设备都不例外。本节将结合具体例子再次审视它们，希望引起足够重视。深入理解 C 语言特性，是编写优质嵌入式 C 程序的基础。

2.1 处处都是陷阱

2.1.1 无心之过

1) “=”和”==”

将比较运算符”==”误写成赋值运算符”=”，可能是绝大多数人都遇到过的，比如下面代码：

```
if(x=5)
{
    //其它代码
}
```

代码的本意是比较变量 x 是否等于常量 5，但是误将”==”写成了”=”，if 语句恒为真。如果在逻辑判断表达式中出现赋值运算符，现在的大多数编译器会给出警告信息。比如 keil MDK 会给出警告提示：“warning: #187-D: use of "=" where"==" may have been intended”，但并非所有程序员都会注意到这类警告，因此有经验的程序员使用下面的代码来避免此类错误：

```
if(5==x)
{
    //其它代码
}
```

将常量放在变量 x 的左边，即使程序员误将’==’写成了’=’，编译器会产生一个任谁也不能无视的语法错误信息：不可给常量赋值！

2) 复合赋值运算符

复合赋值运算符（+=、*=等等）虽然可以使表达式更加简洁并有可能产生更高效的机器代码，但某些复合赋值运算符也会给程序带来隐含 Bug，比如”+=”容易误写成”=+”，代码如下：

tmp=+1;

代码本意是想表达 tmp=tmp+1，但是将复合赋值运算符”+=”误写成”=+”：将正整数常量 1 赋值给变量 tmp。编译器会欣然接受这类代码，连警告都不会产生。

如果你能在调试阶段就发现这个 Bug，真应该庆祝一下，否则这很可能会成为一个重大隐含 Bug，且不易被察觉。

复合赋值运算符”-=”也有类似问题存在。

3) 其它容易误写

使用了中文标点

头文件声明语句最后忘记结束分号

逻辑与&&和位与&、逻辑或||和位或|、逻辑非！和位取反~

字母 l 和数字 1、字母 O 和数字 0

这些误写其实容易被编译器检测出，只需要关注编译器对此的提示信息，就能很快解决。

很多的软件 Bug 源自于输入错误。在 Google 上搜索的时候，有些结果列表项中带有一条警告，表明 Google 认为它带有恶意代码。如果你在 2009 年 1 月 31 日一大早使用 Google 搜索的话，你就会看到，在那天早晨 55 分钟的时间内，Google 的搜索结果标明每个站点对你的 PC 都是有害的。这涉及到整个 Internet 上的所有站点，包括 Google 自己的所有站点和服务。Google 的恶意软件检测功能通过在一个已知攻击者的列表上查找站点，从而识别出危险站点。在 1 月 31 日早晨，对这个列表的更新意外地包含了一条斜杠(“/”)。所有的 URL 都包含一条斜杠，并且，反恶意软件功能把这条斜杠理解为所有的 URL 都是可疑的，因此，它愉快地对搜索结果中的每个站点都添加一条警告。很少见到如此简单的一个输入错误带来的结果如此奇怪且影响如此广泛，但程序就是这样，容不得一丝疏忽。

2.1.2 数组下标

数组常常也是引起程序不稳定的重要因素，C 语言数组的迷惑性与数组下标从 0 开始密不可分，你可以定义 int test[30]，但是你绝不可以使用数组元素 test [30]，除非你自己明确知道在做什么。

2.1.3 容易被忽略的 break 关键字

1) 不能漏加的 break

switch…case 语句可以很方便的实现多分支结构，但要注意在合适的位置添加 break 关键字。程序员往往容易漏加 break 从而引起顺序执行多个 case 语句，这也许是 C 的一个缺陷之处。

对于 switch…case 语句，从概率论上说，绝大多数程序一次只需执行一个匹配的 case 语句，而每一个这样的 case 语句后都必须跟一个 break。去复杂化大概率事件，这多少有些不合常情。

2) 不能乱加的 break

break 关键字用于跳出最近的那层循环语句或者 switch 语句，但程序员往往不够重视这一点。

1990 年 1 月 15 日，AT&T 电话网络位于纽约的一台交换机宕机并且重启，引起它邻近交换机瘫痪，由此及彼，一个连着一个，很快，114 型交换机每六秒宕机重启一次，六万人九小时内不能打长途电话。当时的解决方式：工程师重装了以前的软件版本。。。事后的事故调查发现，这是 break 关键字误用造成的。《C 专家编程》提供了一个简化版的问题源码：

```
network code()
{
    switch(line)
     {
        case   THING1:
           {
            doit1();
         } break;
        case   THING2:
           {
            if(x==STUFF)
             {
                do_first_stuff();
                if(y==OTHER_STUFF)
                    break;
                do_later_stuff();
          }   /*代码的意图是跳转到这里… …*/
            initialize_modes_pointer();
           } break;
        default :
            processing();
    } /*… …但事实上跳到了这里。*/
    use_modes_pointer(); /*致使 modes_pointer 未初始化*/
}
```

那个程序员希望从 if 语句跳出，但他却忘记了 break 关键字实际上跳出最近的那层循环语句或者 switch 语句。现在它跳出了 switch 语句，执行了 use_modes_pointer()函数。但必要的初始化工作并未完成，为将来程序的失败埋下了伏笔。

2.1.4 意想不到的八进制

将一个整形常量赋值给变量，代码如下所示：

```
int a=34, b=034;
```

变量 a 和 b 相等吗？

答案是不相等的。我们知道，16 进制常量以’0x’为前缀，10 进制常量不需要前缀，那么 8 进制呢？它与 10 进制和 16 进制表示方法都不相通，它以数字’0’为前缀，这多少有点奇葩：三种进制的表示方法完全不相通。如果 8 进制也像 16 进制那样以数字和字母表示前缀的话，或许更有利于减少软件 Bug，毕竟你使用 8 进制的次数可能都不会有误使用的次数多！下面展示一个误用 8 进制的例子，最后一个数组元素赋值错误：

```
a[0]=106;          /*十进制数 106*/
a[1]=112;          /*十进制数 112*/
a[2]=052;          /*实际为十进制数 42，本意为十进制 52*/
```

2.1.5 指针加减运算

指针的加减运算是特殊的。下面的代码运行在 32 位 ARM 架构上，执行之后，a 和 p 的值分别是多少？

```
int a=1;
int *p=(int *)0x00001000;
a=a+1;
p=p+1;
```

对于 a 的值很容判断出结果为 2，但是 p 的结果却是 0x00001004。指针 p 加 1 后，p 的值增加了 4，这是为什么呢？原因是指针做加减运算时是以指针的数据类型为单位。p+1 实际上是按照公式 p+1*sizeof(int)来计算的。不理解这一点，在使用指针直接操作数据时极易犯错。

某项目使用下面代码对连续 RAM 初始化零操作，但运行发现有些 RAM 并没有被真正清零。

```
unsigned int *pRAMaddr;                 //定义地址指针变量
for(pRAMaddr=StartAddr;pRAMaddr<EndAddr;pRAMaddr+=4)
{
      *pRAMaddr=0x00000000;     //指定 RAM 地址清零
}
```

通过分析我们发现，由于 pRAMaddr 是一个无符号 int 型指针变量，所以 pRAMaddr+=4 代码其实使 pRAMaddr 偏移了 4*sizeof(int)=16 个字节，所以每执行一次 for 循环，会使变量 pRAMaddr 偏移 16 个字节空间，但只有 4 字节空间被初始化为零。其它的 12 字节数据的内容，在大多数架构处理器中都会是随机数。

2.1.6 关键字 sizeof

不知道有多少人最初认为 sizeof 是一个函数。其实它是一个关键字，其作用是返回一个对象或者类型所占的内存字节数，对绝大多数编译器而言，返回值为无符号整形数据。需要注意的是，使用 sizeof 获取数组长度时，不要对指针应用 sizeof 操作符，比如下面的例子：

```
void ClearRAM(char array[])
{
    int i ;
    for(i=0;i<sizeof(array)/sizeof(array[0]);i++)        //这里用法错误，array 实际上是指针
    {
        array[i]=0x00;
    }
}

int main(void)
{
    char Fle[20];
    ClearRAM(Fle);                //只能清除数组 Fle 中的前四个元素
}
```

我们知道，对于一个数组 array[20]，我们使用代码 sizeof(array)/sizeof(array[0])可以获得数组的元素（这里为 20），但数组名和指针往往是容易混淆的，有且只有一种情况下数组名是可以当做指针的，那就是数组名作为函数形参时，数组名被认为是指针，同时，它不能

再兼任数组名。注意只有这种情况下，数组名才可以当做指针，但不幸的是这种情况下容易引发风险。在 ClearRAM 函数内，作为形参的 array[]不再是数组名了，而成了指针。sizeof(array)相当于求指针变量占用的字节数，在 32 位系统下，该值为 4，sizeof(array)/sizeof(array[0])的运算结果也为 4。所以在 main 函数中调用 ClearRAM(Fle)，也只能清除数组 Fle 中的前四个元素了。

2.1.7 增量运算符’++’和减量运算符’—‘

增量运算符”++”和减量运算符”--“既可以做前缀也可以做后缀。前缀和后缀的区别在于值的增加或减少这一动作发生的时间是不同的。作为前缀是先自加或自减然后做别的运算，作为后缀时，是先做运算，之后再自加或自减。许多程序员对此认识不够，就容易埋下隐患。下面的例子可以很好的解释前缀和后缀的区别。

```
int a=8,b=2,y;
y=a+++--b;
```

代码执行后，y 的值是多少？

这个例子并非是挖空心思设计出来专门让你绞尽脑汁的 C 难题（如果你觉得自己对 C 细节掌握很有信心，做一些 C 难题检验一下是个不错的选择。那么，《The C Puzzle Book》这本书一定不要错过），你甚至可以将这个难懂的语句作为不友好代码的例子。但是它也可以让你更好的理解 C 语言。根据运算符优先级以及编译器识别字符的贪心法原则，第二句代码可以写成更明确的形式：

```
y=(a++)+(--b);
```

当赋值给变量 y 时，a 的值为 8，b 的值为 1,所以变量 y 的值为 9；赋值完成后，变量 a 自加，a 的值变为 9，千万不要以为 y 的值为 10。这条赋值语句相当于下面的两条语句：

```
y=a+(--b);
a=a+1;
```

2.1.8 逻辑与’&&’和逻辑或’||’的陷阱

为了提高系统效率，逻辑与和逻辑或操作的规定如下：如果对第一个操作数求值后就可以推断出最终结果，第二个操作数就不会进行求值！比如下面代码：

```
if((i>=0)&&(i++ <=max))
{
        //其它代码
}
```

在这个代码中，只有当 i>=0 时，i++才会被执行。这样，i 是否自增是不够明确的，这可能会埋下隐患。逻辑或与之类似。

2.1.9 结构体的填充

结构体可能产生填充，因为对大多数处理器而言，访问按字或者半字对齐的数据速度更快，当定义结构体时，编译器为了性能优化，可能会将它们按照半字或字对齐，这样会带来填充问题。比如以下两个个结构体：

第一个结构体：

```
struct {
    char   c；
    short s；
    int    x；
}str_test1;
```

第二个结构体：

```
struct {
    char   c;
    int    x;
    short s;
}str_test2;
```

这两个结构体元素都是相同的变量，只是元素换了下位置，那么这两个结构体变量占用的内存大小相同吗？

其实这两个结构体变量占用的内存是不同的，对于 Keil MDK 编译器，默认情况下第一个结构体变量占用 8 个字节，第二个结构体占用 12 个字节，差别很大。第一个结构体变量在内存中的存储格式如图 2-1 所示：

图 2-1：结构体变量 1 内存分布

第二个结构体变量在内存中的存储格式如图 2-2 所示。对比两个图可以看出 MDK 编译器是是怎么将数据对齐的，这其中的填充内容是之前内存中的数据，是随机的，所以不能再结构之间逐字节比较；另外，合理的排布结构体内的元素位置，可以最大限度减少填充，节省 RAM。

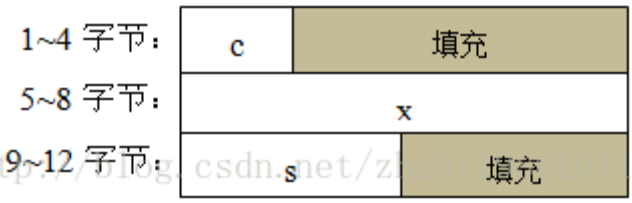


图 2-2 ：结构体变量 2 内存分布

2.2 不可轻视的优先级

C 语言有 32 个关键字，却有 34 个运算符。要记住所有运算符的优先级是困难的。稍不注意，你的代码逻辑和实际执行就会有很大出入。

比如下面将 BCD 码转换为十六进制数的代码：

```
result=(uTimeValue>>4)*10+uTimeValue&0x0F;
```

这里 uTimeValue 存放的 BCD 码，想要转换成 16 进制数据，实际运行发现，如果 uTimeValue 的值为 0x23，按照我设定的逻辑，result 的值应该是 0x17，但运算结果却是 0x07。经过种种排查后，才发现’+’的优先级是大于’&’的，相当于(uTimeValue>>4)*10+uTimeValue 与 0x0F 位与，结果自然与逻辑不符。符合逻辑的代码应该是：

```
result=(uTimeValue>>4)*10+(uTimeValue&0x0F);
```

不合理的#define 会加重优先级问题，让问题变得更加隐蔽。

```
#define READSDA IO0PIN&(1<<11)   //读 IO 口 p0.11 的端口状态

if(READSDA==(1<<11))              //判断端口 p0.11 是否为高电平
{
    //其它代码
}
```

编译器在编译后将宏带入,原代码语句变为:

```
if(IO0PIN&(1<<11) ==(1<<11))
{
    //其它代码
}
```

运算符'=='的优先级是大于'&'的，代码 IO0PIN&(1<<11) ==(1<<11)) 等效为 IO0PIN&0x00000001：判断端口 P0.0 是否为高电平，这与原意相差甚远。因此，使用宏定义的时候，最好将被定义的内容用括号括起来。

按照常规方式使用时，可能引起误会的运算符还有很多，如表 2-1 所示。C 语言的运算符当然不会只止步于数目繁多！

有一个简便方法可以避免优先级问题：不清楚的优先级就加上”()”，但这样至少有会带来两个问题：

过多的括号影响代码的可读性，包括自己和以后的维护人员

别人的代码不一定用括号来解决优先级问题，但你总要读别人的代码

无论如何，在嵌入式编程方面，该掌握的基础知识，偷巧不得。建议花一些时间，将优先级顺序以及容易出错的优先级运算符理清几遍。

2.3 隐式转换

C 语言的设计理念一直被人吐槽，因为它认为 C 程序员完全清楚自己在做什么，其中一个证据就是隐式转换。C 语言规定，不同类型的数据（比如 char 和 int 型数据）需要转换成同一类型后，才可进行计算。如果你混合使用类型，比如用 char 类型数据和 int 类型数据做减法，C 使用一个规则集合来自动（隐式的）完成类型转换。这可能很方便，但也很危险。

这就要求我们理解这个转换规则并且能应用到程序中去！

1) 当出现在表达式里时,有符号和无符号的 char 和 short 类型都将自动被转换为 int 类型，在需要的情况下，将自动被转换为 unsigned int（在 short 和 int 具有相同大小时）。这称为类型提升。

提升在算数运算中通常不会有什么大的坏处,但如果位运算符 ~ 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 unsigned char 或者 unsigned short 类型（取决于操作时使用的类型）。

```
uint8_t   port =0x5aU;
uint8_t   result_8;
result_8= (~port) >> 4;
```

假如我们不了解表达式里的类型提升，认为在运算过程中变量 port 一直是 unsigned char 类型的。我们来看一下运算过程：~port 结果为 0xa5，0xa5>>4 结果为 0x0a，这是我们期望的值。但实际上，result_8 的结果却是 0xfa！在 ARM 结构下，int 类型为 32 位。变量 port 在运算前被提升为 int 类型：~port 结果为 0xffffffa5，0xa5>>4 结果为 0x0fffffa，赋值给变量 result_8，发生类型截断（这也是隐式的！），result_8=0xfa。经过这么诡异的隐式转换，结果跟我们期望的值，已经大相径庭！正确的表达式语句应该为：

```
result_8=(unsigned char) (~port) >> 4;                    /*强制转换*/
```

2) 在包含两种数据类型的任何运算里，两个值都会被转换成两种类型里较高的级别。类型级别从高到低的顺序是 long double、double、float、unsigned long long、long long、unsigned long、long、unsigned int、int。

这种类型提升通常都是件好事，但往往有很多程序员不能真正理解这句话，比如下面的例子（int 类型表示 16 位）。

```
uint16_t    u16a = 40000;                    /* 16 位无符号变量*/
uint16_t    u16b= 30000;                  /*16 位无符号变量*/
uint32_t    u32x;                         /*32 位无符号变量 */
uint32_t    u32y;
u32x = u16a +u16b;                        /* u32x = 70000 还是 4464 ? */
u32y =(uint32_t)(u16a + u16b);            /* u32y = 70000  还是 4464 ? */
```

u32x 和 u32y 的结果都是 4464（70000%65536）!不要认为表达式中有一个高类别 uint32_t 类型变量，编译器都会帮你把所有其他低类别都提升到 uint32_t 类型。正确的书写方式：

```
u32x = (uint32_t)u16a +(uint32_t)u16b;          或者：
u32x = (uint32_t)u16a + u16b;
```

后一种写法在本表达式中是正确的，但是在其它表达式中不一定正确，比如：

```
uint16_t u16a,u16b,u16c;
uint32_t    u32x;
u32x= u16a + u16b + (uint32_t)u16c;/*错误写法，u16a+ u16b 仍可能溢出*/
```

3) 在赋值语句里，计算的最后结果被转换成将要被赋予值的那个变量的类型。这一过程可能导致类型提升也可能导致类型降级。降级可能会导致问题。比如将运算结果为 321 的值赋值给 8 位 char 类型变量。程序必须对运算时的数据溢出做合理的处理。很多其他语言，像 Pascal（C 语言设计者之一曾撰文狠狠批评过 Pascal 语言），都不允许混合使用类型，但 C 语言不会限制你的自由，即便这经常引起 Bug。

4) 当作为函数的参数被传递时，char 和 short 会被转换为 int，float 会被转换为 double。

当不得已混合使用类型时，一个比较好的习惯是使用类型强制转换。强制类型转换可以避免编译器隐式转换带来的错误，同时也向以后的维护人员传递一些有用信息。这有个前提：你要对强制类型转换有足够的了解！下面总结一些规则：

并非所有强制类型转换都是由风险的，把一个整数值转换为一种具有相同符号的更宽类型时，是绝对安全的。

精度高的类型强制转换为精度低的类型时，通过丢弃适当数量的最高有效位来获取结果，也就是说会发生数据截断，并且可能改变数据的符号位。

精度低的类型强制转换为精度高的类型时，如果两种类型具有相同的符号，那么没什么问题；需要注意的是负的有符号精度低类型强制转换为无符号精度高类型时，会不直观的执行符号扩展，例如：

```
unsigned int bob;
signed char fred = -1;
bob=(unsigned int )fred;                  /*发生符号扩展，此时 bob 为 0xFFFFFFFF*/
```

3.编译器

如果你和一个优秀的程序员共事，你会发现他对他使用的工具非常熟悉，就像一个画家了解他的画具一样。----比尔.盖茨

3.1 不能简单的认为是个工具

嵌入式程序开发跟硬件密切相关，需要使用 C 语言来读写底层寄存器、存取数据、控制硬件等，C 语言和硬件之间由编译器来联系，一些 C 标准不支持的硬件特性操作，由编译器提供。

汇编可以很轻易的读写指定 RAM 地址、可以将代码段放入指定的 Flash 地址、可以精确的设置变量在 RAM 中分布等等，所有这些操作，在深入了解编译器后，也可以使用 C 语

言实现。

C 语言标准并非完美，有着数目繁多的未定义行为，这些未定义行为完全由编译器自主决定，了解你所用的编译器对这些未定义行为的处理，是必要的。

嵌入式编译器对调试做了优化，会提供一些工具，可以分析代码性能，查看外设组件等，了解编译器的这些特性有助于提高在线调试的效率。

此外，堆栈操作、代码优化、数据类型的范围等等，都是要深入了解编译器的理由。

如果之前你认为编译器只是个工具，能够编译就好。那么，是时候改变这种思想了。

3.2 不能依赖编译器的语义检查

编译器的语义检查很弱小，甚至还会“掩盖”错误。现代的编译器设计是件浩瀚的工程，为了让编译器设计简单一些，目前几乎所有编译器的语义检查都比较弱小。为了获得更快的执行效率，C 语言被设计的足够灵活且几乎不进行任何运行时检查，比如数组越界、指针是否合法、运算结果是否溢出等等。这就造成了很多编译正确但执行奇怪的程序。

C 语言足够灵活，对于一个数组 test[30]，它允许使用像 test[-1]这样的形式来快速获取数组首元素所在地址前面的数据；允许将一个常数强制转换为函数指针，使用代码(*((void(*)())0))()来调用位于 0 地址的函数。C 语言给了程序员足够的自由，但也由程序员承担滥用自由带来的责任。

3.2.1 莫名的死机

下面的两个例子都是死循环，如果在不常用分支中出现类似代码，将会造成看似莫名其妙的死机或者重启。

```
1.  unsigned char i;        //例程 1
2.  for(i=0;i<256;i++)
3.  {
4.      //其它代码
5.  }
1.  unsigned char i;        //例程 2
2.  for(i=10;i>=0;i--)
3.  {
4.      //其它代码
5.  }
```

对于无符号 char 类型，表示的范围为 0~255，所以无符号 char 类型变量 i 永远小于 256（第一个 for 循环无限执行），永远大于等于 0（第二个 for 循环无线执行）。需要说明的是，赋值代码 i=256 是被 C 语言允许的，即使这个初值已经超出了变量 i 可以表示的范围。C 语言会千方百计的为程序员创造出错的机会，可见一斑。

3.2.2 不起眼的改变

假如你在 if 语句后误加了一个分号，可能会完全改变了程序逻辑。编译器也会很配合的帮忙掩盖，甚至连警告都不提示。代码如下：

```
1.  if(a>b);                //这里误加了一个分号
2.  a=b;                    //这句代码一直被执行
```

不但如此，编译器还会忽略掉多余的空格符和换行符，就像下面的代码也不会给出足

够提示：

```
1.  if(n<3)
2.  return              //这里少加了一个分号
3.  logrec.data=x[0];
4.  logrec.time=x[1];
5.  logrec.code=x[2];
```

这段代码的本意是 n<3 时程序直接返回，由于程序员的失误，return 少了一个结束分号。编译器将它翻译成返回表达式 logrec.data=x[0]的结果，return 后面即使是一个表达式也是 C 语言允许的。这样当 n>=3 时，表达式 logrec.data=x[0];就不会被执行，给程序埋下了隐患。

3.2.3 难查的数组越界

上文曾提到数组常常是引起程序不稳定的重要因素，程序员往往不经意间就会写数组越界。

一位同事的代码在硬件上运行，一段时间后就会发现 LCD 显示屏上的一个数字不正常的被改变。经过一段时间的调试，问题被定位到下面的一段代码中：

```
1.  int SensorData[30];
2.  //其他代码
3.  for(i=30;i>0;i--)
4.  {
5.      SensorData[i]=…;
6.      //其他代码
7.  }
```

这里声明了拥有 30 个元素的数组，不幸的是 for 循环代码中误用了本不存在的数组元素 SensorData[30]，但 C 语言却默许这么使用，并欣然的按照代码改变了数组元素 SensorData[30]所在位置的值， SensorData[30]所在的位置原本是一个 LCD 显示变量，这正是显示屏上的那个值不正常被改变的原因。真庆幸这么轻而易举的发现了这个 Bug。

其实很多编译器会对上述代码产生一个警告：赋值超出数组界限。但并非所有程序员都对编译器警告保持足够敏感，况且，编译器也并不能检查出数组越界的所有情况。比如下面的例子：

你在模块 A 中定义数组：

```
1.  int SensorData[30];
```

在模块 B 中引用该数组，但由于你引用代码并不规范，这里没有显示声明数组大小，但编译器也允许这么做：

```
1.  extern int SensorData[];
```

这次，编译器不会给出警告信息，因为编译器压根就不知道数组的元素个数。所以，当一个数组声明为具有外部链接，它的大小应该显式声明。

再举一个编译器检查不出数组越界的例子。函数 func()的形参是一个数组形式，函数代码简化如下所示：

```
1.  char * func(char SensorData[30])
2.  {
3.      unsignedint i；
4.      for(i=30;i>0;i--)
5.      {
6.          SensorData[i]=…;
7.          //其他代码
8.      }
9.  }
```

这个给 SensorData[30]赋初值的语句，编译器也是不给任何警告的。实际上，编译器是将数组名 Sensor 隐含的转化为指向数组第一个元素的指针，函数体是使用指针的形式来访问数组的，它当然也不会知道数组元素的个数了。造成这种局面的原因之一是 C 编译器的作者们认为指针代替数组可以提高程序效率，而且，可以简化编译器的复杂度。

指针和数组是容易给程序造成混乱的，我们有必要仔细的区分它们的不同。其实换一个角度想想，它们也是容易区分的：可以将数组名等同于指针的情况有且只有一处，就是上面例子提到的数组作为函数形参时。其它时候，数组名是数组名，指针是指针。

下面的例子编译器同样检查不出数组越界。

我们常常用数组来缓存通讯中的一帧数据。在通讯中断中将接收的数据保存到数组中，直到一帧数据完全接收后再进行处理。即使定义的数组长度足够长，接收数据的过程中也可能发生数组越界，特别是干扰严重时。这是由于外界的干扰破坏了数据帧的某些位，对一帧的数据长度判断错误，接收的数据超出数组范围，多余的数据改写与数组相邻的变量，造成系统崩溃。由于中断事件的异步性，这类数组越界编译器无法检查到。

如果局部数组越界，可能引发 ARM 架构硬件异常。

同事的一个设备用于接收无线传感器的数据，一次软件升级后，发现接收设备工作一段时间后会死机。调试表明 ARM7 处理器发生了硬件异常，异常处理代码是一段死循环（死机的直接原因）。接收设备有一个硬件模块用于接收无线传感器的整包数据并存在自己的缓冲区中，当硬件模块接收数据完成后，使用外部中断通知设备取数据，外部中断服务程序精简后如下所示：

```
1.  __irq ExintHandler(void)
2.  {
3.      unsignedchar DataBuf[50];
4.      GetData(DataBug);          //从硬件缓冲区取一帧数据
5.      //其他代码
6.  }
```

由于存在多个无线传感器近乎同时发送数据的可能加之 GetData()函数保护力度不够，数组 DataBuf 在取数据过程中发生越界。由于数组 DataBuf 为局部变量，被分配在堆栈中，同在此堆栈中的还有中断发生时的运行环境以及中断返回地址。溢出的数据将这些数据破坏掉，中断返回时 PC 指针可能变成一个不合法值，硬件异常由此产生。

如果我们精心设计溢出部分的数据，化数据为指令，就可以利用数组越界来修改 PC 指针的值，使之指向我们希望执行的代码。

1988 年，第一个网络蠕虫在一天之内感染了 2000 到 6000 台计算机，这个蠕虫程序利用的正是一个标准输入库函数的数组越界 Bug。起因是一个标准输入输出库函数 gets()，原来设计为从数据流中获取一段文本，遗憾的是，gets()函数没有规定输入文本的长度。gets()函数内部定义了一个 500 字节的数组，攻击者发送了大于 500 字节的数据，利用溢出的数据修改了堆栈中的 PC 指针，从而获取了系统权限。目前，虽然有更好的库函数来代替 gets 函数，但 gets 函数仍然存在着。

3.2.4 神奇的 volatile

做嵌入式设备开发，如果不对 volatile 修饰符具有足够了解，实在是说不过去。volatile 是 C 语言 32 个关键字中的一个，属于类型限定符，常用的 const 关键字也属于类型限定符。

volatile 限定符用来告诉编译器，该对象的值无任何持久性，不要对它进行任何优化；它迫使编译器每次需要该对象数据内容时都必须读该对象，而不是只读一次数据并将它放在寄存器中以便后续访问之用（这样的优化可以提高系统速度）。

这个特性在嵌入式应用中很有用，比如你的 IO 口的数据不知道什么时候就会改变，这就要求编译器每次都必须真正的读取该 IO 端口。这里使用了词语“真正的读”，是因为由于编译器的优化，你的逻辑反应到代码上是对的，但是代码经过编译器翻译后，有可能与你的逻辑不符。你的代码逻辑可能是每次都会读取 IO 端口数据，但实际上编译器将代码翻译成汇编时，可能只是读一次 IO 端口数据并保存到寄存器中，接下来的多次读 IO 口都是使用寄存器中的值来进行处理。因为读写寄存器是最快的，这样可以优化程序效率。与之类似的，中断里的变量、多线程中的共享变量等都存在这样的问题。

不使用 volatile，可能造成运行逻辑错误，但是不必要的使用 volatile 会造成代码效率低下（编译器不优化 volatile 限定的变量），因此清楚的知道何处该使用 volatile 限定符，是一个嵌入式程序员的必修内容。

一个程序模块通常由两个文件组成，源文件和头文件。如果你在源文件定义变量：

```
1.  unsigned int test;
```

并在头文件中声明该变量：

```
1.  extern unsigned long test;
```

编译器会提示一个语法错误：变量’ test’声明类型不一致。但如果你在源文件定义变量：

```
1.  volatile unsigned int test;
```
在头文件中这样声明变量：

```
1.  extern unsigned int test;        /*缺少 volatile 限定符*/
```
编译器却不会给出错误信息（有些编译器仅给出一条警告）。当你在另外一个模块（该模块包含声明变量 test 的头文件）使用变量 test 时，它已经不再具有 volatile 限定，这样很可能造成一些重大错误。比如下面的例子，注意该例子是为了说明 volatile 限定符而专门构造出的，因为现实中的 volatile 使用 Bug 大都隐含，并且难以理解。

在模块 A 的源文件中，定义变量：

```
1.  volatile unsigned int TimerCount=0;
```
该变量用来在一个定时器中断服务程序中进行软件计时：

```
1.  TimerCount++;
```
在模块 A 的头文件中，声明变量：

```
1.  extern unsigned int TimerCount;    //这里漏掉了类型限定符 volatile
```
在模块 B 中，要使用 TimerCount 变量进行精确的软件延时：

```
1.  #include “…A.h”                          //首先包含模块 A 的头文件
2.  //其他代码
3.  TimerCount=0;
4.  while(TimerCount<=TIMER_VALUE);    //延时一段时间(感谢网友 chhfish 指出这里的逻辑错误)
5.  //其他代码
```
实际上，这是一个死循环。由于模块 A 头文件中声明变量 TimerCount 时漏掉了 volatile 限定符，在模块 B 中，变量 TimerCount 是被当作 unsigned int 类型变量。由于寄存器速度远快于 RAM，编译器在使用非 volatile 限定变量时是先将变量从 RAM 中拷贝到寄存器中，如果同一个代码块再次用到该变量，就不再从 RAM 中拷贝数据而是直接使用之前寄存器备份值。代码 while(TimerCount<=TIMER_VALUE)中，变量 TimerCount 仅第一次执行时被使用，之后都是使用的寄存器备份值，而这个寄存器值一直为 0，所以程序无限循环。图 3-1 的流程图说明了程序使用限定符 volatile 和不使用 volatile 的执行过程。

为了更容易的理解编译器如何处理 volatile 限定符，这里给出未使用 volatile 限定符和使用 volatile 限定符程序的反汇编代码：

没有使用关键字 volatile，在 keil MDK V4.54 下编译，默认优化级别，如下所示（注意最后两行）：

```
122:        unIdleCount=0;
2.          123:
3.  0x00002E10  E59F11D4  LDR          R1,[PC,#0x01D4]
4.  0x00002E14  E3A05000  MOV          R5,#key1(0x00000000)
5.  0x00002E18  E1A00005  MOV          R0,R5
```

```
6.  0x00002E1C  E5815000  STR       R5,[R1]
7.      124:      while(unIdleCount!=200);   //延时 2S 钟
8.      125:
9.       0x00002E20  E35000C8  CMP       R0,#0x000000C8
10. 0x00002E24  1AFFFFFD  BNE       0x00002E20</span>
```

使用关键字 volatile，在 keil MDK V4.54 下编译，默认优化级别，如下所示（注意最后三行）：

```
122:      unIdleCount=0;
2.      123:
3.  0x00002E10  E59F01D4  LDR       R0,[PC,#0x01D4]
4.  0x00002E14  E3A05000  MOV       R5,#key1(0x00000000)
5.  0x00002E18  E5805000  STR       R5,[R0]
6.      124:      while(unIdleCount!=200);   //延时 2S 钟
7.      125:
8.  0x00002E1C  E5901000  LDR       R1,[R0]
9.  0x00002E20  E35100C8  CMP       R1,#0x000000C8
10. 0x00002E24  1AFFFFFC  BNE       0x00002E1C
```

可以看到，如果没有使用 volatile 关键字，程序一直比较 R0 内数据与 0xC8 是否相等，但 R0 中的数据是 0，所以程序会一直在这里循环比较（死循环）；再看使用了 volatile 关键字的反汇编代码，程序会先从变量中读出数据放到 R1 寄存器中，然后再让 R1 内数据与 0xC8 相比较，这才是我们 C 代码的正确逻辑！

3.2.5 局部变量

ARM 架构下的编译器会频繁的使用堆栈，堆栈用于存储函数的返回值、AAPCS 规定的必须保护的寄存器以及局部变量，包括局部数组、结构体、联合体和 C++的类。默认情况下，堆栈的位置、初始值都是由编译器设置，因此需要对编译器的堆栈有一定了解。从堆栈中分配的局部变量的初值是不确定的，因此需要运行时显式初始化该变量。一旦离开局部变量的作用域，这个变量立即被释放，其它代码也就可以使用它，因此堆栈中的一个内存位置可能对应整个程序的多个变量。

局部变量必须显式初始化，除非你确定知道你要做什么。下面的代码得到的温度值跟预期会有很大差别，因为在使用局部变量 sum 时，并不能保证它的初值为 0。编译器会在第一次运行时清零堆栈区域，这加重了此类 Bug 的隐蔽性。

```
1.  unsigned intGetTempValue(void)
2.  {
3.      unsigned int sum;                    //定义局部变量，保存总值
4.      for(i=0;i<10;i++)
5.      {
6.          sum+=CollectTemp();              //函数 CollectTemp 可以得到当前的温度值
7.      }
8.      return (sum/10);
9.  }
```

由于一旦程序离开局部变量的作用域即被释放，所以下面代码返回指向局部变量的指

针是没有实际意义的，该指针指向的区域可能会被其它程序使用，其值会被改变。

```
1.  char * GetData(void)
2.  {
3.      char buffer[100];                    //局部数组
4.      …
5.      return buffer;
6.  }
```

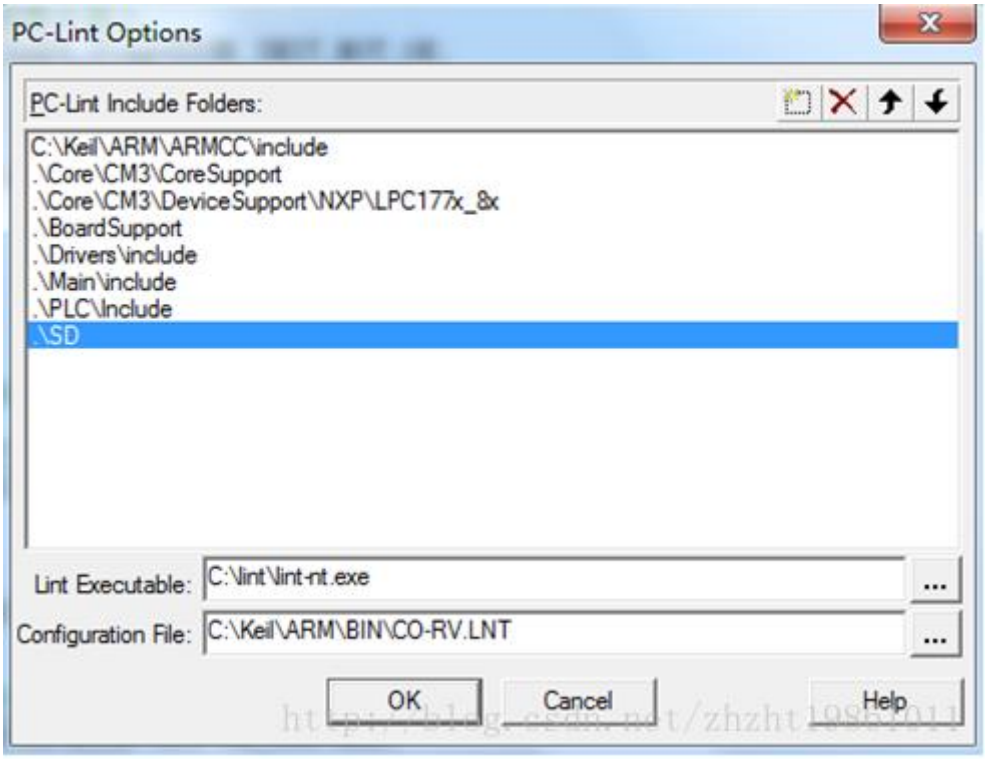
3.2.6 使用外部工具

由于编译器的语义检查比较弱，我们可以使用第三方代码分析工具，使用这些工具来发现潜在的问题，这里介绍其中比较著名的是 PC-Lint。

PC-Lint 由 Gimpel Software 公司开发，可以检查 C 代码的语法和语义并给出潜在的 BUG 报告。PC-Lint 可以显著降低调试时间。

目前公司 ARM7 和 Cortex-M3 内核多是使用 Keil MDK 编译器来开发程序，通过简单配置，PC-Lint 可以被集成到 MDK 上，以便更方便的检查代码。MDK 已经提供了 PC-Lint 的配置模板，所以整个配置过程十分简单，Keil MDK 开发套件并不包含 PC-Lint 程序，在此之前，需要预先安装可用的 PC-Lint 程序，配置过程如下：

1) 点击菜单 Tools---Set-up PC-Lint…



PC-Lint Include Folders：该列表路径下的文件才会被 PC-Lint 检查，此外，这些路径下的文件内使用#include 包含的文件也会被检查；

Lint Executable：指定 PC-Lint 程序的路径

Configuration File：指定配置文件的路径，该配置文件由 MDK 编译器提供。

2) 菜单 Tools---Lint 文件路径.c/.h

检查当前文件。

3) 菜单 Tools---Lint All C-Source Files

检查所有 C 源文件。

PC-Lint 的输出信息显示在 MDK 编译器的 Build Output 窗口中，双击其中的一条信息可以跳转到源文件所在位置。

编译器语义检查的弱小在很大程度上助长了不可靠代码的广泛存在。随着时代的进步，现在越来越多的编译器开发商意识到了语义检查的重要性，编译器的语义检查也越来越强大，比如公司使用的 Keil MDK 编译器，虽然它的编辑器依然不尽人意，但在其 V4.47 及以上版本中增加了动态语法检查并加强了语义检查，可以友好的提示更多警告信息。建议经常关注编译器官方网站并将编译器升级到 V4.47 或以上版本，升级的另一个好处是这些版本的编辑器增加了标识符自动补全功能，可以大大节省编码的时间。

3.3 你觉得有意义的代码未必正确

C 语言标准特别的规定某些行为是未定义的，编写未定义行为的代码，其输出结果由编译器决定！ C 标准委员会定义未定义行为的原因如下：

简化标准，并给予实现一定的灵活性，比如不捕捉那些难以诊断的程序错误；
编译器开发商可以通过未定义行为对语言进行扩展

C 语言的未定义行为，使得 C 极度高效灵活并且给编译器实现带来了方便，但这并不利于优质嵌入式 C 程序的编写。因为许多 C 语言中看起来有意义的东西都是未定义的，并且这也容易使你的代码埋下隐患，并且不利于跨编译器移植。Java 程序会极力避免未定义行为，并用一系列手段进行运行时检查，使用 Java 可以相对容易的写出安全代码，但体积庞大效率低下。作为嵌入式程序员，我们需要了解这些未定义行为，利用 C 语言的灵活性，写出比 Java 更安全、效率更高的代码来。

3.3.1 常见的未定义行为

1) 自增自减在表达式中连续出现并作用于同一变量或者自增自减在表达式中出现一次，但作用的变量多次出现

自增（++）和自减（--）这一动作发生在表达式的哪个时刻是由编译器决定的，比如：

```
1.  r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
```

不同的编译器可能有着不同的汇编代码，可能是先执行 i++再进行乘法和加法运行，也可能是先进行加法和乘法运算，再执行 i++，因为这句代码在一个表达式中出现了连续的自增并作用于同一变量。更加隐蔽的是自增自减在表达式中出现一次，但作用的变量多次出现，比如：

```
1.  a[i] = i++; /*  未定义行为  */
```

先执行 i++再赋值，还是先赋值再执行 i++是由编译器决定的，而两种不同的执行顺序的结果差别是巨大的。

2) 函数实参被求值的顺序

函数如果有多个实参，这些实参的求值顺序是由编译器决定的，比如：

```
1.  printf("%d %d\n", ++n, power(2, n));      /*  未定义行为  */
```

是先执行++n 还是先执行 power(2,n)是由编译器决定的。

3) 有符号整数溢出

有符号整数溢出是未定义的行为，编译器决定有符号整数溢出按照哪种方式取值。比如下面代码：

```
1.  int value1,value2,sum
2.
3.  //其它操作
4.  sum=value1+value;      /*sum 可能发生溢出*/
```

4) 有符号数右移、移位的数量是负值或者大于操作数的位数

5) 除数为零

6) malloc()、calloc()或 realloc()分配零字节内存

3.3.2 如何避免 C 语言未定义行为

代码中引入未定义行为会为代码埋下隐患，防止代码中出现未定义行为是困难的，我们总能不经意间就会在代码中引入未定义行为。但是还是有一些方法可以降低这种事件，总结如下：

了解 C 语言未定义行为

标准 C99 附录 J.2“未定义行为”列举了 C99 中的显式未定义行为，通过查看该文档，了解那些行为是未定义的，并在编码中时刻保持警惕；

寻求工具帮助

编译器警告信息以及 PC-Lint 等静态检查工具能够发现很多未定义行为并警告，要时

刻关注这些工具反馈的信息；

总结并使用一些编码标准

1）避免构造复杂的自增或者自减表达式，实际上，应该避免构造所有复杂表达式；

比如 a[i] = i++;语句可以改为 a[i] = i; i++;这两句代码。

2）只对无符号操作数使用位操作；

必要的运行时检查

检查是否溢出、除数是否为零，申请的内存数量是否为零等等，比如上面的有符号整数溢出例子，可以按照如下方式编写，以消除未定义特性：

```
1.  int value1,value2,sum;
2.
3.  //其它代码
4.  if((value1>0 && value2>0 && value1>(INT_MAX-value2))||
5.      (value1<0 && value2<0 && value1<(INT_MIN-value2)))
6.  {
7.      //处理错误
8.  }
9.  else
10. {
11.     sum=value1+value2;
12. }
```

上面的代码是通用的，不依赖于任何 CPU 架构，但是代码效率很低。如果是有符号数使用补码的 CPU 架构（目前常见 CPU 绝大多数都是使用补码），还可以用下面的代码来做溢出检查：

```
int value1, value2, sum;
unsigned int usum = (unsigned int)value1 + value2;

if((usum ^ value1) & (usum ^ value2) & INT_MIN)
{
    /*处理溢出情况*/
}
else
{
    sum = value1 + value2;
}
```

使用的原理解释一下，因为在加法运算中，操作数 value1 和 value2 只有符号相同时，才可能发生溢出，所以我们先将这两个数转换为无符号类型，两个数的和保存在变量 usum 中。如果发生溢出，则 value1、value2 和 usum 的最高位（符号位）一定不同，表达式(usum ^ value1) & (usum ^ value2) 的最高位一定为 1，这个表达式位与（&）上 INT_MIN 是为了将最高位之

外的其它位设置为 0。

了解你所用的编译器对未定义行为的处理策略

很多引入了未定义行为的程序也能运行良好，这要归功于编译器处理未定义行为的策略。不是你的代码写的正确，而是恰好编译器处理策略跟你需要的逻辑相同。了解编译器的未定义行为处理策略，可以让你更清楚的认识到那些引入了未定义行为程序能够运行良好是多么幸运的事，不然多换几个编译器试试！

以 Keil MDK 为例，列举常用的处理策略如下：

1） 有符号量的右移是算术移位，即移位时要保证符号位不改变。

2）对于 int 类的值：超过 31 位的左移结果为零；无符号值或正的有符号值超过 31 位的右移结果为零。负的有符号值移位结果为-1。

3）整型数除以零返回零

3.4 了解你的编译器

在嵌入式开发过程中，我们需要经常和编译器打交道，只有深入了解编译器，才能用好它，编写更高效代码，更灵活的操作硬件，实现一些高级功能。下面以公司最常用的 Keil MDK 为例，来描述一下编译器的细节。

3.4.1 编译器的一些小知识

1) 默认情况下，char 类型的数据项是无符号的，所以它的取值范围是 0～255；

2) 在所有的内部和外部标识符中，大写和小写字符不同；

3) 通常局部变量保存在寄存器中，但当局部变量太多放到栈里的时候，它们总是字对齐的。

4) 压缩类型的自然对齐方式为 1。使用关键字__packed 来压缩特定结构，将所有有效类型的对齐边界设置为 1；

5) 整数以二进制补码形式表示；浮点量按 IEEE 格式存储；

6) 整数除法的余数的符号于被除数相同，由 ISO C90 标准得出；

7) 如果整型值被截断为短的有符号整型，则通过放弃适当数目的最高有效位来得到结果。如果原始数是太大的正或负数，对于新的类型，无法保证结果的符号将于原始数相同。

8) 整型数超界不引发异常；像 unsigned char test; test=1000;这类是不会报错的；

9) 在严格 C 中，枚举值必须被表示为整型。例如，必须在-2147483648 到+2147483647 的范围内。但 MDK 自动使用对象包含 enum 范围的最小整型来实现（比如 char 类型），除

非使用编译器命令--enum_is_int 来强制将 enum 的基础类型设为至少和整型一样宽。超出范围的枚举值默认仅产生警告：#66:enumeration value is out of "int" range；

10) 对于结构体填充，根据定义结构的方式，keil MDK 编译器用以下方式的一种来填充结构：

I> 定义为 static 或者 extern 的结构用零填充；

II> 栈或堆上的结构，例如，用 malloc()或者 auto 定义的结构，使用先前存储在那些存储器位置的任何内容进行填充。不能使用 memcmp()来比较以这种方式定义的填充结构！

11) 编译器不对声明为 volatile 类型的数据进行优化；

12) __nop()：延时一个指令周期，编译器绝不会优化它。如果硬件支持 NOP 指令，则该句被替换为 NOP 指令，如果硬件不支持 NOP 指令，编译器将它替换为一个等效于 NOP 的指令，具体指令由编译器自己决定；

13) __align(n)：指示编译器在 n 字节边界上对齐变量。对于局部变量，n 的值为 1、2、4、8；

14) __attribute__((at(address)))：可以使用此变量属性指定变量的绝对地址；

15) __inline：提示编译器在合理的情况下内联编译 C 或 C++ 函数；

3.4.2 初始化的全局变量和静态变量的初始值被放到了哪里？

我们程序中的一些全局变量和静态变量在定义时进行了初始化，经过编译器编译后，这些初始值被存放在了代码的哪里？我们举个例子说明：

```
1. unsigned int g_unRunFlag=0xA5；
2. static unsigned int s_unCountFlag=0x5A；
```

我曾做过一个项目，项目中的一个设备需要在线编程，也就是通过协议，将上位机发给设备的数据通过在应用编程（IAP）技术写入到设备的内部 Flash 中。我将内部 Flash 做了划分，一小部分运行程序，大部分用来存储上位机发来的数据。随着程序量的增加，在一次更新程序后发现，在线编程之后，设备运行正常，但是重启设备后，运行出现了故障！经过一系列排查，发现故障的原因是一个全局变量的初值被改变了。这是件很不可思议的事情，你在定义这个变量的时候指定了初始值，当你在第一次使用这个变量时却发现这个初值已经被改掉了！这中间没有对这个变量做任何赋值操作，其它变量也没有任何溢出，并且多次在线调试表明，进入 main 函数的时候，该变量的初值已经被改为一个恒定值。

要想知道为什么全局变量的初值被改变，就要了解这些初值编译后被放到了二进制文件的哪里。在此之前，需要先了解一点链接原理。

ARM 映象文件各组成部分在存储系统中的地址有两种：一种是映象文件位于存储器时（通俗的说就是存储在 Flash 中的二进制代码）的地址，称为加载地址；一种是映象文件运

行时（通俗的说就是给板子上电，开始运行 Flash 中的程序了）的地址，称为运行时地址。赋初值的全局变量和静态变量在程序还没运行的时候，初值是被放在 Flash 中的，这个时候他们的地址称为加载地址，当程序运行后，这些初值会从 Flash 中拷贝到 RAM 中，这时候就是运行时地址了。

原来，对于在程序中赋初值的全局变量和静态变量，程序编译后，MDK 将这些初值放到 Flash 中，位于紧靠在可执行代码的后面。在程序进入 main 函数前，会运行一段库代码，将这部分数据拷贝至相应 RAM 位置。由于我的设备程序量不断增加，超过了为设备程序预留的 Flash 空间，在线编程时，将一部分存储全局变量和静态变量初值的 Flash 给重新编程了。在重启设备前，初值已经被拷贝到 RAM 中，所以这个时候程序运行是正常的，但重新上电后，这部分初值实际上是在线编程的数据，自然与初值不同了。

3.4.3 在 C 代码中使用的变量，编译器将他们分配到 RAM 的哪里？

我们会在代码中使用各种变量，比如全局变量、静态变量、局部变量，并且这些变量时由编译器统一管理的，有时候我们需要知道变量用掉了多少 RAM，以及这些变量在 RAM 中的具体位置。这是一个经常会遇到的事情，举一个例子，程序中的一个变量在运行时总是不正常的被改变，那么有理由怀疑它临近的变量或数组溢出了，溢出的数据更改了这个变量值。要排查掉这个可能性，就必须知道该变量被分配到 RAM 的哪里、这个位置附近是什么变量，以便针对性的做跟踪。

其实 MDK 编译器的输出文件中有一个“工程名.map”文件，里面记录了代码、变量、堆栈的存储位置，通过这个文件，可以查看使用的变量被分配到 RAM 的哪个位置。要生成这个文件，需要在 Options for Targer 窗口，Listing 标签栏下，勾选 Linker Listing 前的复选框，如图 3-1 所示。

图 3-1 设置编译器生产 MAP 文件

3.4.4 默认情况下，栈被分配到 RAM 的哪个地方？

MDK 中，我们只需要在配置文件中定义堆栈大小，编译器会自动在 RAM 的空闲区域选择一块合适的地方来分配给我们定义的堆栈，这个地方位于 RAM 的那个地方呢？

通过查看 MAP 文件，原来 MDK 将堆栈放到程序使用到的 RAM 空间的后面，比如你的 RAM 空间从 0x4000 0000 开始，你的程序用掉了 0x200 字节 RAM，那么堆栈空间就从 0x4000 0200 处开始。

使用了多少堆栈，是否溢出?

2.4.5 有多少 RAM 会被初始化？

在进入 main()函数之前，MDK 会把未初始化的 RAM 给清零的，我们的 RAM 可能很大，只使用了其中一小部分，MDK 会不会把所有 RAM 都初始化呢？

答案是否定的，MDK 只是把你的程序用到的 RAM 以及堆栈 RAM 给初始化，其它 RAM

的内容是不管的。如果你要使用绝对地址访问 MDK 未初始化的 RAM，那就要小心翼翼的了，因为这些 RAM 上电时的内容很可能是随机的，每次上电都不同。

3.4.6 MDK 编译器如何设置非零初始化变量？

对于控制类产品，当系统复位后（非上电复位），可能要求保持住复位前 RAM 中的数据，用来快速恢复现场，或者不至于因瞬间复位而重启现场设备。而 keil mdk 在默认情况下，任何形式的复位都会将 RAM 区的非初始化变量数据清零。

MDK 编译程序生成的可执行文件中，每个输出段都最多有三个属性：RO 属性、RW 属性和 ZI 属性。对于一个全局变量或静态变量，用 const 修饰符修饰的变量最可能放在 RO 属性区，初始化的变量会放在 RW 属性区，那么剩下的变量就要放到 ZI 属性区了。默认情况下，ZI 属性区的数据在每次复位后，程序执行 main 函数内的代码之前，由编译器“自作主张”的初始化为零。所以我们要在 C 代码中设置一些变量在复位后不被零初始化，那一定不能任由编译器“胡作非为”，我们要用一些规则，约束一下编译器。

分散加载文件对于连接器来说至关重要，在分散加载文件中，使用 UNINIT 来修饰一个执行节，可以避免编译器对该区节的 ZI 数据进行零初始化。这是要解决非零初始化变量的关键。因此我们可以定义一个 UNINIT 修饰的数据节，然后将希望非零初始化的变量放入这个区域中。于是，就有了第一种方法：

1) 修改分散加载文件，增加一个名为 MYRAM 的执行节，该执行节起始地址为 0x1000A000，长度为 0x2000 字节（8KB），由 UNINIT 修饰：

```
1:    LR_IROM1 0x00000000 0x00080000  {      ; load region size_region
2:    ER_IROM1 0x00000000 0x00080000  {  ; load address = execution address
3:      *.o (RESET, +First)
4:      *(InRoot$$Sections)
5:      .ANY (+RO)
6:    }
7:    RW_IRAM1 0x10000000 0x0000A000  {  ; RW data
8:      .ANY (+RW +ZI)
9:    }
10:    MYRAM 0x1000A000 UNINIT 0x00002000  {
11:      .ANY (NO_INIT)
12:    }
13: }
```

那么，如果在程序中有一个数组，你不想让它复位后零初始化，就可以这样来定义变量：

```
1.  unsigned char  plc_eu_backup[32] __attribute__((at(0x1000A000)));
```

变量属性修饰符__attribute__((at(adde)))用来将变量强制定位到 adde 所在地址处。由于地址 0x1000A000 开始的 8KB 区域 ZI 变量不会被零初始化，所以位于这一区域的数组 plc_eu_backup 也就不会被零初始化了。

这种方法的缺点是显而易见的：要程序员手动分配变量的地址。如果非零初始化数据比较多，这将是件难以想象的大工程（以后的维护、增加、修改代码等等）。所以要找到一种办法，让编译器去自动分配这一区域的变量。

2) 分散加载文件同方法 1，如果还是定义一个数组，可以用下面方法：

```
unsigned char    plc_eu_backup[32] __attribute__((section("NO_INIT"),zero_init));
```

变量属性修饰符__attribute__((section(“name”),zero_init))用于将变量强制定义到 name 属性数据节中，zero_init 表示将未初始化的变量放到 ZI 数据节中。因为“NO_INIT”这显性命名的自定义节，具有 UNINIT 属性。

3) 将一个模块内的非初始化变量都非零初始化

假如该模块名字为 test.c，修改分散加载文件如下所示：

```
1: LR_IROM1 0x00000000 0x00080000  {       ; load region size_region
2:   ER_IROM1 0x00000000 0x00080000  {  ; load address = execution address
3:     *.o (RESET, +First)
4:     *(InRoot$$Sections)
5:     .ANY (+RO)
6:   }
7:   RW_IRAM1 0x10000000 0x0000A000  {  ; RW data
8:     .ANY (+RW +ZI)
9:   }
10:   RW_IRAM2 0x1000A000 UNINIT 0x00002000  {
11:     test.o (+ZI)
12:   }
13: }
```

在该模块定义时变量时使用如下方法：

这里，变量属性修饰符__attribute__((zero_init))用于将未初始化的变量放到 ZI 数据节中变量，其实 MDK 默认情况下，未初始化的变量就是放在 ZI 数据区的。

4.防御性编程

嵌入式产品的可靠性自然与硬件密不可分，但在硬件确定、并且没有第三方测试的前提下，使用防御性编程思想写出的代码，往往具有更高的稳定性。

防御性编程首先需要认清 C 语言的种种缺陷和陷阱，C 语言对于运行时的检查十分弱小，需要程序员谨慎的考虑代码，在必要的时候增加判断；防御性编程的另一个核心思想是假设代码运行在并不可靠的硬件上，外接干扰有可能会打乱程序执行顺序、更改 RAM 存储数据等等。

4.1 具有形参的函数，需判断传递来的实参是否合法。

程序员可能无意识的传递了错误参数；外界的强干扰可能将传递的参数修改掉，或者使用随机参数意外的调用函数，因此在执行函数主体前，需要先确定实参是否合法。

```
1.  int exam_fun( unsigned char *str )
2.  {
3.          if( str != NULL )       //  检查“假设指针不为空”这个条件
4.              {
5.                  //正常处理代码
6.          }
7.          else
8.          {
9.                  //处理错误代码
10.       }
11. }
```

4.2 仔细检查函数的返回值

对函数返回的错误码，要进行全面仔细处理，必要时做错误记录。

```
1.  char *DoSomething(…)
2.  {
3.      char * p;
4.        p=malloc(1024);
5.        if(p==NULL)                /*对函数返回值作出判断*/
6.        {
7.            UARTprintf(…);     /*打印错误信息*/
8.             return NULL;
9.        }
10.       retuen p;
11. }
```

4.3 防止指针越界

如果动态计算一个地址时，要保证被计算的地址是合理的并指向某个有意义的地方。特别对于指向一个结构或数组的内部的指针，当指针增加或者改变后仍然指向同一个结构或数组。

4.4 防止数组越界

数组越界的问题前文已经讲述的很多了，由于 C 不会对数组进行有效的检测，因此必须在应用中显式的检测数组越界问题。下面的例子可用于中断接收通讯数据。

```
1.  #define REC_BUF_LEN 100
2.  unsigned char RecBuf[REC_BUF_LEN];
3.  //其它代码
4.  void Uart_IRQHandler(void)
5.  {
6.      static RecCount=0;              //接收数据长度计数器
```

```
7.      //其它代码
8.      if(RecCount< REC_BUF_LEN)     //判断数组是否越界
9.      {
10.             RecBuf[RecCount]=…;       //从硬件取数据
11.             RecCount++;
12.             //其它代码
13.     }
14.     else
15.     {
16.         //错误处理代码
17.     }
18.      //其它代码
19. }
```

在使用一些库函数时，同样需要对边界进行检查，比如下面的 memset(RecBuf,0,len)函数把 RecBuf 指指向的内存区的前 len 个字节用 0 填充，如果不注意 len 的长度，就会将数组 RecBuf 之外的内存区清零：

```
1.  #define REC_BUF_LEN 100
2.  unsigned char RecBuf[REC_BUF_LEN];
3.
4.  if(len< REC_BUF_LEN)
5.  {
6.      memset(RecBuf,0,len);          //将数组 RecBuf 清零
7.  }
8.  else
9.  {
10.     //处理错误
11. }
```

4.5 数学算数运算

4.5.1 除法运算，只检测除数为零就可靠吗？

除法运算前，检查除数是否为零几乎已经成为共识，但是仅检查除数是否为零就够了吗？

考虑两个整数相除，对于一个 signed long 类型变量，它能表示的数值范围为：-2147483648 ~+2147483647，如果让-2147483648/ -1，那么结果应该是+2147483648，但是这个结果已经超出了 signedlong 所能表示的范围了。所以，在这种情况下，除了要检测除数是否为零外，还要检测除法是否溢出。

```
1.  #include <limits.h>
2.  signed long sl1,sl2,result;
3.  /*初始化 sl1 和 sl2*/
4.  if((sl2==0)||(sl1==LONG_MIN && sl2==-1))
```

```
5.  {
6.      //处理错误
7.  }
8.  else
9.  {
10.     result = sl1 / sl2;
11. }
```

4.5.2 检测运算溢出

整数的加减乘运算都有可能发生溢出，在讨论未定义行为时，给出过一个有符号整形加法溢出判断代码，这里再给出一个无符号整形加法溢出判断代码段：

```
1.  #include <limits.h>
2.  unsigned int a,b,result;
3.  /*初始化 a，b*/
4.  if(UINT_MAX-a<b)
5.  {
6.      //处理溢出
7.  }
8.  else
9.  {
10.     result=a+b;
11. }
```

嵌入式硬件一般没有浮点处理器，浮点数运算在嵌入式也比较少见并且溢出判断严重依赖 C 库支持，这里不讨论。

4.5.3 检测移位

在讨论未定义行为时，提到有符号数右移、移位的数量是负值或者大于操作数的位数都是未定义行为，也提到不对有符号数进行位操作，但要检测移位的数量是否大于操作数的位数。下面给出一个无符号整数左移检测代码段：

```
1.  unsigned int ui1;
2.  unsigned int ui2;
3.  unsigned int uresult;
4.
5.  /*初始化 ui1,ui2*/
6.  if(ui2>=sizeof(unsigned int)*CHAR_BIT)
7.  {
8.      //处理错误
9.  }
10. else
11. {
12.     uresult=ui1<<ui2;
13. }
```

4.6 如果有硬件看门狗，则使用它

在其它一切措施都失效的情况下，看门狗可能是最后的防线。它的原理特别简单，但却能大大提高设备的可靠性。如果设备有硬件看门狗，一定要为它编写驱动程序。

要尽可能早的开启看门狗

这是因为从上电复位结束到开启看门狗的这段时间内，设备有可能被干扰而跳过看门狗初始化程序，导致看门狗失效。尽可能早的开启看门狗，可以降低这种概率；

不要在中断中喂狗，除非有其他联动措施

在中断程序喂狗，由于干扰的存在，程序可能一直处于中断之中，这样会导致看门狗失效。如果在主程序中设置标志位，中断程序喂狗时与这个标志位联合判断，也是允许的；

喂狗间隔跟产品需求有关，并非特定的时间

产品的特性决定了喂狗间隔。对于不涉及安全性、实时性的设备，喂狗间隔比较宽松，但间隔时间不宜过长，否则被用户感知到，是影响用户体验的。对于设计安全性、有实时控制类的设备，原则是尽可能快的复位，否则会造成事故。

克莱门汀号在进行第二阶段的任务时，原本预订要从月球飞行到太空深处的 Geographos 小行星进行探勘，然而这艘太空探测器在飞向小行星时却由于一个软件缺陷而使其中断运作 20 分钟，不但未能到达小行星，也因为控制喷嘴燃烧了 11 分钟使电力供应降低，无法再透过远端控制探测器，最终结束这项任务，但也导致了资源与资金的浪费。

“克莱门汀太空任务失败这件事让我感到十分震惊，它其实可以透过硬件中一款简单的看门狗计时器避免掉这项意外，但由于当时的开发时间相当紧缩，程序设计人员没时间编写程序来启动它，”Ganssle 说。

遗憾的是，1998 年发射的近地号太空船(NEAR)也遇到了相同的问题。由于编程人员并未采纳建议，因此，当推进器减速器系统故障时，29 公斤的储备燃料也随之报销——这同样是一个本来可经由看门狗定时器编程而避免的问题，同时也证明要从其他程序设计人员的错误中学习并不容易。

4.7 关键数据储存多个备份，取数据采用“表决法”

RAM 中的数据在受到干扰情况下有可能被改变，对于系统关键数据应该进行保护。关键数据包括全局变量、静态变量以及需要保护的数据区域。备份数据与原数据不应该处于相邻位置，因此不应由编译器默认分配备份数据位置，而应该由程序员指定区域存储。可以将 RAM 分为 3 个区域，第一个区域保存原码，第二个区域保存反码，第三个区域保存异或码，区域之间预留一定量的“空白”RAM 作为隔离。可以使用编译器的“分散加载”机制将变量分别存储在这些区域。需要进行读取时，同时读出 3 份数据并进行表决，取至少有两个相同的那个值。

假如设备的 RAM 从 0x1000_0000 开始，我需要在 RAM 的 0x1000_0000~0x10007FFF 内存储原码，在 0x1000_9000~0x10009FFF 内存储反码，在 0x1000_B000~0x1000BFFF 内存储 0xAA 的异或码，编译器的分散加载可以设置为：

```
1.  LR_IROM1 0x00000000 0x00080000  {    ; load region size_region
```

```
2.      ER_IROM1 0x00000000 0x00080000   {   ; load address = execution address
3.        *.o (RESET, +First)
4.        *(InRoot$$Sections)
5.        .ANY (+RO)
6.      }
7.      RW_IRAM1 0x10000000 0x00008000   {   ;保存原码
8.        .ANY (+RW +ZI )
9.      }
10.
11.     RW_IRAM3 0x10009000 0x00001000{      ;保存反码
12.       .ANY (MY_BK1)
13.     }
14.
15.     RW_IRAM2 0x1000B000 0x00001000   {   ;保存异或码
16.       .ANY (MY_BK2)
17.     }
18. }
```

如果一个关键变量需要多处备份，可以按照下面方式定义变量，将三个变量分别指定到三个不连续的 RAM 区中，并在定义时按照原码、反码、0xAA 的异或码进行初始化。

```
1.   uint32   plc_pc=0;                                                    //原码
2.   __attribute__((section("MY_BK1"))) uint32 plc_pc_not=~0x0;           //反码
3.   __attribute__((section("MY_BK2"))) uint32 plc_pc_xor=0x0^0xAAAAAAAA;  //异或码
```

当需要写这个变量时，这三个位置都要更新；读取变量时，读取三个值做判断，取至少有两个相同的那个值。

为什么选取异或码而不是补码？这是因为 MDK 的整数是按照补码存储的，正数的补码与原码相同，在这种情况下，原码和补码是一致的，不但起不到冗余作用，反而对可靠性有害。比如存储的一个非零整数区因为干扰，RAM 都被清零，由于原码和补码一致，按照 3 取 2 的“表决法”，会将干扰值 0 当做正确的数据。

4.8 对非易失性存储器进行备份存储

非易失性存储器包括但不限于 Flash、EEPROM、铁电。仅仅将写入非易失性存储器中的数据再读出校验是不够的。强干扰情况下可能导致非易失性存储器内的数据错误，在写非易失性存储器的期间系统掉电将导致数据丢失，因干扰导致程序跑飞到写非易失性存储器函数中，将导致数据存储紊乱。一种可靠的办法是将非易失性存储器分成多个区，每个数据都将按照不同的形式写入到这些分区中，需要进行读取时，同时读出多份数据并进行表决，取相同数目较多的那个值。

4.9 软件锁

对于初始化序列或者有一定先后顺序的函数调用，为了保证调用顺序或者确保每个函

数都被调用，我们可以使用环环相扣，实质上这也是一种软件锁。此外对于一些安全关键代码语句（是语句，而不是函数），可以给它们设置软件锁，只有持有特定钥匙的，才可以访问这些关键代码。也可以通俗的理解为，关键安全代码不能按照单一条件执行，要额外的多设置一个标志。

比如，向 Flash 写一个数据，我们会判断数据是否合法、写入的地址是否合法，计算要写入的扇区。之后调用写 Flash 子程序，在这个子程序中，判断扇区地址是否合法、数据长度是否合法，之后就要将数据写入 Flash。由于写 Flash 语句是安全关键代码，所以程序给这些语句上锁：必须具有正确的钥匙才可以写 Flash。这样即使是程序跑飞到写 Flash 子程序，也能大大降低误写的风险。

```
1.
    /*****************************************************************************
**
2.  * 名称：RamToFlash()
3.  * 功能：复制 RAM 的数据到 FLASH，命令代码 51。
4.  * 入口参数： dst          目标地址，即 FLASH 起始地址。以 512 字节为分界
5.  *           src          源地址，即 RAM 地址。地址必须字对齐
6.  *           no           复制字节个数，为 512/1024/4096/8192
7.  *           ProgStart    软件锁标志
8.  * 出口参数：IAP 返回值 (paramout 缓冲区) CMD_SUCCESS,SRC_ADDR_ERROR,DST_ADDR_ERROR,
9.  SRC_ADDR_NOT_MAPPED,DST_ADDR_NOT_MAPPED,COUNT_ERROR,BUSY,未选择扇区
10.
    *****************************************************************************
*/
11. void   RamToFlash(uint32 dst, uint32 src, uint32 no,uint8 ProgStart)
12. {
13.     PLC_ASSERT("Sector number",(dst>=0x00040000)&&(dst<=0x0007FFFF));
14.     PLC_ASSERT("Copy bytes number is 512",(no==512));
15.     PLC_ASSERT("ProgStart==0xA5",(ProgStart==0xA5));
16.
17.     paramin[0] = IAP_RAMTOFLASH;                  // 设置命令字
18.     paramin[1] = dst;                            // 设置参数
19.     paramin[2] = src;
20.     paramin[3] = no;
21.     paramin[4] = Fcclk/1000;
22.     if(ProgStart==0xA5)                          //只有软件锁标志正确时,才执行关键代码
23.     {
24.         iap_entry(paramin, paramout);          // 调用 IAP 服务程序
25.         ProgStart=0;
26.     }
27.     else
```

```
28.     {
29.         paramout[0]=PROG_UNSTART;
30.     }
31. }
```

该程序段是编程 lpc1778 内部 Flash，其中调用 IAP 程序的函数 iap_entry(paramin, paramout)是关键安全代码，所以在执行该代码前，先判断一个特定设置的安全锁标志 ProgStart，只有这个标志符合设定值，才会执行编程 Flash 操作。如果因为意外程序跑飞到该函数，由于 ProgStart 标志不正确，是不会对 Flash 进行编程的。

4.10 通信

通讯线上的数据误码相对严重，通讯线越长，所处的环境越恶劣，误码会越严重。抛开硬件和环境的作用，我们的软件应能识别错误的通讯数据。对此有一些应用措施：

制定协议时，限制每帧的字节数；

每帧字节数越多，发生误码的可能性就越大，无效的数据也会越多。对此以太网规定每帧数据不大于 1500 字节，高可靠性的 CAN 收发器规定每帧数据不得多于 8 字节，对于 RS485，基于 RS485 链路应用最广泛的 Modbus 协议一帧数据规定不超过 256 字节。因此，建议制定内部通讯协议时，使用 RS485 时规定每帧数据不超过 256 字节；

使用多种校验

编写程序时应使能奇偶校验，每帧超过 16 字节的应用，建议至少编写 CRC16 校验程序；

增加额外判断

1)增加缓冲区溢出判断。这是因为数据接收多是在中断中完成，编译器检测不出缓冲区是否溢出，需要手动检查，在上文介绍数据溢出一节中已经详细说明。

2)增加超时判断。当一帧数据接收到一半，长时间接收不到剩余数据，则认为这帧数据无效，重新开始接收。可选，跟不同的协议有关，但缓冲区溢出判断必须实现。这是因为对于需要帧头判断的协议，上位机可能发送完帧头后突然断电，重启后上位机是从新的帧开始发送的，但是下位机已经接收到了上次未发送完的帧头，所以上位机的这次帧头会被下位机当成正常数据接收。这有可能造成数据长度字段为一个很大的值，填满该长度的缓冲区需要相当多的数据（比如一帧可能 1000 字节），影响响应时间；另一方面，如果程序没有缓冲区溢出判断，那么缓冲区很可能溢出，后果是灾难性的。

重传机制

如果检测到通讯数据发生了错误，则要有重传机制重新发送出错的帧。

4.11 开关量输入的检测、确认

开关量容易受到尖脉冲干扰，如果不进行滤除，可能会造成误动作。一般情况下，需要对开关量输入信号进行多次采样，并进行逻辑判断直到确认信号无误为止。

4.12 开关量输出

开关信号简单的一次输出是不安全的，干扰信号可能会翻转开关量输出的状态。采取

重复刷新输出可以有效防止电平的翻转。

4.13 初始化信息的保存和恢复

微处理器的寄存器值也可能会因外界干扰而改变，外设初始化值需要在寄存器中长期保存，最容易被破坏。由于 Flash 中的数据相对不易被破坏，可以将初始化信息预先写入 Flash，待程序空闲时比较与初始化相关的寄存器值是否被更改，如果发现非法更改则使用 Flash 中的值进行恢复。

公司目前使用的 4.3 寸 LCD 显示屏抗干扰能力一般。如果显示屏与控制器之间的排线距离过长或者对使用该显示屏的设备打静电或者脉冲群，显示屏有可能会花屏或者白屏。对此，我们可以将初始化显示屏的数据保存在 Flash 中，程序运行后，每隔一段时间从显示屏的寄存器读出当前值和 Flash 存储的值相比较，如果发现两者不同，则重新初始化显示屏。下面给出校验源码，仅供参考。

定义数据结构：

```
1.  typedef struct {
2.      uint8_t  lcd_command;          //LCD 寄存器
3.      uint8_t  lcd_get_value[8];     //初始化时写入寄存器的值
4.      uint8_t  lcd_value_num;        //初始化时写入寄存器值的数目
5.  }lcd_redu_list_struct;
```

定义 const 修饰的结构体变量，存储 LCD 部分寄存器的初始值，这个初始值跟具体的应用初始化有关，不一定是表中的数据，通常情况下，这个结构体变量被存储到 Flash 中。

```
1.  /*LCD 部分寄存器设置值列表*/
2.  lcd_redu_list_struct const lcd_redu_list_str[]=
3.  {
4.      {SSD1963_Get_Address_Mode,{0x20}                                    ,1},
/*1*/
5.      {SSD1963_Get_Pll_Mn       ,{0x3b,0x02,0x04}                         ,3}, /*2*/
6.      {SSD1963_Get_Pll_Status   ,{0x04}                                   ,1}, /*3*/
7.      {SSD1963_Get_Lcd_Mode     ,{0x24,0x20,0x01,0xdf,0x01,0x0f,0x00}     ,7}, /*4*/
8.      {SSD1963_Get_Hori_Period ,{0x02,0x0c,0x00,0x2a,0x07,0x00,0x00,0x00},8}, /*5*/
9.      {SSD1963_Get_Vert_Period ,{0x01,0x1d,0x00,0x0b,0x09,0x00,0x00}     ,7}, /*6*/
10.     {SSD1963_Get_Power_Mode   ,{0x1c}                                   ,1},
/*7*/
11.     {SSD1963_Get_Display_Mode,{0x03}                                    ,1}, /*8*/
12.     {SSD1963_Get_Gpio_Conf    ,{0x0F,0x01}                              ,2}, /*9*/
13.     {SSD1963_Get_Lshift_Freq ,{0x00,0xb8}                               ,2}, /*10*/
14. };
```

实现函数如下所示，函数会遍历结构体变量中的每一个命令，以及每一个命令下的初始值，如果有一个不正确，则跳出循环，执行重新初始化和恢复措施。这个函数中的 MY_DEBUGF 宏是我自己的调试函数，使用串口打印调试信息，在接下来的第五部分将详细叙述。通过这个函数，我可以长时间监控显示屏的哪些命令、哪些位容易被干扰。程序里使用了一个被妖

魔化的关键字：goto。大多数 C 语言书籍对 goto 关键字谈之色变，但你应该有自己的判断。在函数内部跳出多重循环，除了 goto 关键字，又有哪种方法能如此简洁高效！

```
/**
* lcd 显示冗余
*  每隔一段时间调用该程序一次
*/
void lcd_redu(void)
{
    uint8_t   tmp[8];
    uint32_t i,j;
    uint32_t lcd_init_flag;

    lcd_init_flag =0;
    for(i=0;i<sizeof(lcd_redu_list_str)/sizeof(lcd_redu_list_str[0]);i++)
    {
        LCD_SendCommand(lcd_redu_list_str[i].lcd_command);
        uyDelay(10);
        for(j=0;j<lcd_redu_list_str[i].lcd_value_num;j++)
        {
            tmp[j]=LCD_ReadData();
            if(tmp[j]!=lcd_redu_list_str[i].lcd_get_value[j])
            {
                lcd_init_flag=0x55;
                MY_DEBUGF(MENU_DEBUG,("读 lcd 寄存器值与预期不符,命令为:0x%x,第%d 个参数,
                该参数正确值为:0x%x, 实际读出值为:0x%x\n",lcd_redu_list_str[i].lcd_command,j+1,
                lcd_redu_list_str[i].lcd_get_value[j],tmp[j]));
                goto handle_lcd_init;
            }
        }
    }

    handle_lcd_init:
    if(lcd_init_flag==0x55)
    {
        //重新初始化 LCD
        //一些必要的恢复措施
    }
}
```

4.14 陷阱

对于 8051 内核单片机，由于没有相应的硬件支持，可以用纯软件设置软件陷阱，用来拦截一些程序跑飞。对于 ARM7 或者 Cortex-M 系列单片机，硬件已经内建了多种异常，软

件需要根据硬件异常来编写陷阱程序，用来快速定位甚至恢复错误。

4.15 阻塞处理

有时候程序员会使用 while(!flag);语句阻塞在此等待标志 flag 改变，比如串口发送时用来等待一字节数据发送完成。这样的代码时存在风险的，如果因为某些原因标志位一直不改变则会造成系统死机。

一个良好冗余的程序是设置一个超时定时器，超过一定时间后，强制程序退出 while 循环。

2003 年 8 月 11 日发生的 W32.Blaster.Worm 蠕虫事件导致全球经济损失高达 5 亿美元，这个漏洞是利用了 Windows 分布式组件对象模型的远程过程调用接口中的一个逻辑缺陷：在调用 GetMachineName()函数时，循环只设置了一个不充分的结束条件。

原代码简化如下所示：

```
1.  HRESULT GetMachineName ( WCHAR *pwszPath,
2.  WCHARwszMachineName[MAX_COMPUTTERNAME_LENGTH_FQDN+1])
3.  {
4.          WCHAR *pwszServerName = wszMachineName;
5.          WCHAR *pwszTemp = pwszPath + 2;
6.          while ( *pwszTemp != L’\\’ )            /* 这句代码循环结束条件不充分 */
7.                  *pwszServerName++= *pwszTemp++;
8.          /*… */
9.  }
```

微软发布的安全补丁 MS03-026 解决了这个问题，为 GetMachineName()函数设置了充分终止条件。一个解决代码简化如下所示（并非微软补丁代码）：

```
1.  HRESULT GetMachineName( WCHAR *pwszPath,
2.  WCHARwszMachineName[MAX_COMPUTTERNAME_LENGTH_FQDN+1])
3.  {
4.          WCHAR *pwszServerName = wszMachineName;
5.          WCHAR *pwszTemp = pwszPath + 2;
6.          WCHAR *end_addr = pwszServerName +MAX_COMPUTTERNAME_LENGTH_FQDN；
7.          while (（*pwszTemp != L’\\’ ) && (*pwszTemp != L’\0’)
8.  && (pwszServerName<end_addr))   /*充分终止条件*/
9.                  *pwszServerName++= *pwszTemp++;
10.         /*… */
11. }
```

5.测试，再测试

思维再缜密的程序员也不可能编写完全无缺陷的程序，测试的目的正是尽可能多的发现这些缺陷并改正。这里说的测试，是指程序员的自测试。前期的自测试能够更早的发现错误，相应的修复成本也会很低，如果你不彻底测试自己的代码，恐怕你开发的就不只是代码，可能还会声名狼藉。

优质嵌入式 C 程序跟优质的基础元素关系密切，可以将函数作为基础元素，我们的测试正是从最基本的函数开始。判断哪些函数需要测试需要一定的经验积累，虽然代码行数跟逻辑复杂度并不成正比，但如果你不能判断某个函数是否要测试，一个简单粗暴的方法是：当函数有效代码超过 20 行，就测试它。

程序员对自己的代码以及逻辑关系十分清楚，测试时，按照每一个逻辑分支全面测试。很多错误发生在我们认为不会出错的地方，所以即便某个逻辑分支很简单，也建议测试一遍。第一个原因是我们自己看自己的代码总是不容易发现错误，而测试能暴露这些错误；另一方面，语法正确、逻辑正确的代码，经过编译器编译后，生成的汇编代码很可能与你的逻辑相差甚远。比如我们前文提及的使用 volatile 以及不使用 volatile 关键字编译后生成的汇编代码，再比如我们用低优化级别编译和使用高优化级别编译后生成的汇编代码，都可能相差很大，实际运行测试，可以暴漏这些隐含错误。最后，虽然可能性极小，编译器本身也可能有 BUG，特别是构造复杂表达式的情况下（应极力避免复杂表达式）。

5.1 使用硬件调试器测试

使用硬件调试器（比如 J-link）测试是最通用的手段。可以单步运行、设置断点，可以很方便的查看当前寄存器、变量的值。在寻找缺陷方面，使用硬件调试器测试是最简单却又最有效的手段。

硬件调试器已经在公司普遍使用，这方面的测试不做介绍，想必大家都已经很熟悉了。

5.2 有些缺陷很难缠

就像没有一种方法能完美解决所有问题，在实际项目中，硬件调试器也有难以触及的地方。可以举几个例子说明：

使用了比较大的协议栈，需要跟进到协议栈内部调试的缺陷

比如公司使用 lwIP 协议栈，如果跟踪数据的处理过程，需要从接收数据开始一直到应用层处理数据，之间会经过驱动层、IP 层、TCP 层和应用层，会经过十几个文件几十个函数，使用硬件调试器跟踪费时费力；

具有随机性的缺陷

有一些缺陷，可能是不定时出现的，有可能是几分钟出现，也有可能是几个小时甚至几天才出现，像这样的缺陷很难用硬件调试器捕捉到；

需要外界一系列有时间限制的输入条件触发，但这一过程中有缺陷

比如我们用组合键来完成某个功能，规定按下按键 1 不小于 3 秒后松开，然后在 6 秒内分别按下按键 2、按键 3、按键 4 这三个按键来执行我们的特定程序，要测试类似这种过程，硬件调试器很难做到；

除了测试缺陷需要，有时候我们在做稳定性测试时，需要知道软件每时每刻运行到那些分支、执行了哪些操作、我们关心的变量当前值是什么等等，这些都表明，我们还需要一种和硬件调试器互补的测试手段。

这个测试手段就是在程序中增加额外调试语句，当程序运行时，通过这些调试语句将运行信息输出到可以方便查看的设备上，可以是 PC 机、LCD 显示屏、存储卡等等。

以串口输出到 PC 机为例，下面提供完整的测试思路。在此之前，我们先对这种测试手段提一些要求：

必须简单易用

我们在初学 C 语言的时候，都接触过 printf 函数，这个函数可以方便的输出信息，并可以将各种变量格式化为指定格式的字符串，我们应当提供类似的函数；

调试语句必须方便的从代码中移除

在编码阶段，我们可能会往程序中加入大量的调试语句，但是程序发布时，需要将这些调试语句从代码中移除，这将是件恐怖的过程。我们必须提供一种策略，可以方便的移除这些调试语句。

5.2.1 简单易用的调试函数

1) 使用库函数 printf。以 MDK 为例，方法如下：

I>初始化串口

II>重构 fputc 函数，printf 函数会调用 fputc 函数执行底层串口的数据发送。

```
/**
  * @brief  将 C 库中的 printf 函数重定向到指定的串口.
  * @param  ch:要发送的字符
  * @param  f :文件指针
  */
int fputc(int ch, FILE *f)
{

  /*这里是一个跟硬件相关函数,将一个字符写到 UART */
  //举例:USART_SendData(UART_COM1, (uint8_t) ch);

  return ch;
}
```

III> 在 Options for Targer 窗口，Targer 标签栏下，勾选 Use MicroLIB 前的复选框以便避免使用半主机功能。（注：标准 C 库 printf 函数默认开启半主机功能，如果非要使用标准 C 库，请自行查阅资料）

2) 构建自己的调试函数

使用库函数比较方便，但也少了一些灵活性，不利于随心所欲的定制输出格式。自己编写类似 printf 函数则会更灵活一些，而且不依赖任何编译器。下面给出一个完整的类 printf 函数实现，该函数支持有限的格式参数，使用方法与库函数一致。同库函数类似，该也需要

提供一个底层串口发送函数（原型为：int32_t UARTwrite(const uint8_t *pcBuf, uint32_t ulLen)），用来发送指定数目的字符，并返回最终发送的字符个数。

```
#include <stdarg.h>                    /*支持函数接收不定量参数*/

const char * const g_pcHex = "0123456789abcdef";

/**
* 简介:    一个简单的 printf 函数,支持\%c, \%d, \%p, \%s, \%u,\%x, and \%X.
*/
void UARTprintf(const uint8_t *pcString, ...)
{
    uint32_t ulIdx;
    uint32_t ulValue;          //保存从不定量参数堆栈中取出的数值型变量
    uint32_t ulPos, ulCount;
    uint32_t ulBase;           //保存进制基数,如十进制则为 10,十六进制数则为 16
    uint32_t ulNeg;            //为 1 表示从变量为负数
    uint8_t *pcStr;            //保存从不定量参数堆栈中取出的字符型变量
    uint8_t pcBuf[32];         //保存数值型变量字符化后的字符
    uint8_t cFill;             //'%08x'->不足 8 个字符用'0'填充,cFill='0';
                                    //'%8x '->不足 8 个字符用空格填充,cFill=' '
    va_list vaArgP;

    va_start(vaArgP, pcString);
    while(*pcString)
    {
        // 首先搜寻非%核字符串结束字符
        for(ulIdx = 0; (pcString[ulIdx] != '%') && (pcString[ulIdx] != '\0'); ulIdx++)
        { }
        UARTwrite(pcString, ulIdx);

        pcString += ulIdx;
        if(*pcString == '%')
        {
            pcString++;

            ulCount = 0;
            cFill = ' ';
again:
            switch(*pcString++)
            {
                case '0': case '1': case '2': case '3': case '4':
                case '5': case '6': case '7': case '8': case '9':
                {
```

```
42.                     // 如果第一个数字为 0, 则使用 0 做填充,则用空格填充)
43.                     if((pcString[-1] == '0') && (ulCount == 0))
44.                     {
45.                         cFill = '0';
46.                     }
47.                     ulCount *= 10;
48.                     ulCount += pcString[-1] - '0';
49.                     goto again;
50.                 }
51.                 case 'c':
52.                 {
53.                     ulValue = va_arg(vaArgP, unsigned long);
54.                     UARTwrite((unsigned char *)&ulValue, 1);
55.                     break;
56.                 }
57.                 case 'd':
58.                 {
59.                     ulValue = va_arg(vaArgP, unsigned long);
60.                     ulPos = 0;
61.
62.                     if((long)ulValue < 0)
63.                     {
64.                         ulValue = -(long)ulValue;
65.                         ulNeg = 1;
66.                     }
67.                     else
68.                     {
69.                         ulNeg = 0;
70.                     }
71.                     ulBase = 10;
72.                     goto convert;
73.                 }
74.                 case 's':
75.                 {
76.                     pcStr = va_arg(vaArgP, unsigned char *);
77.
78.                     for(ulIdx = 0; pcStr[ulIdx] != '\0'; ulIdx++)
79.                     {
80.                     }
81.                     UARTwrite(pcStr, ulIdx);
82.
83.                     if(ulCount > ulIdx)
84.                     {
85.                         ulCount -= ulIdx;
```

```
86.                                     while(ulCount--)
87.                                     {
88.                                         UARTwrite(" ", 1);
89.                                     }
90.                                 }
91.                                 break;
92.                             }
93.                             case 'u':
94.                             {
95.                                 ulValue = va_arg(vaArgP, unsigned long);
96.                                 ulPos = 0;
97.                                 ulBase = 10;
98.                                 ulNeg = 0;
99.                                 goto convert;
100.                            }
101.                            case 'x': case 'X': case 'p':
102.                            {
103.                                ulValue = va_arg(vaArgP, unsigned long);
104.                                ulPos = 0;
105.                                ulBase = 16;
106.                                ulNeg = 0;
107.                convert:    //将数值转换成字符
108.                                for(ulIdx = 1; (((ulIdx * ulBase) <= ulValue) &&(((ulIdx * ulBase) /
ulBase) == ulIdx)); ulIdx *= ulBase, ulCount--)
109.                                { }
110.                                if(ulNeg)
111.                                {
112.                                    ulCount--;
113.                                }
114.                                if(ulNeg && (cFill == '0'))
115.                                {
116.                                    pcBuf[ulPos++] = '-';
117.                                    ulNeg = 0;
118.                                }
119.                                if((ulCount > 1) && (ulCount < 16))
120.                                {
121.                                    for(ulCount--; ulCount; ulCount--)
122.                                    {
123.                                        pcBuf[ulPos++] = cFill;
124.                                    }
125.                                }
126.
127.                                if(ulNeg)
128.                                {
```

```
                                pcBuf[ulPos++] = '-';
                            }

                            for(; ulIdx; ulIdx /= ulBase)
                            {
                                pcBuf[ulPos++] = g_pcHex[(ulValue / ulIdx) % ulBase];
                            }
                            UARTwrite(pcBuf, ulPos);
                            break;
                        }
                        case '%':
                        {
                            UARTwrite(pcString - 1, 1);
                            break;
                        }
                        default:
                        {
                            UARTwrite("ERROR", 5);
                            break;
                        }
                    }
                }
            }
            //可变参数处理结束
            va_end(vaArgP);
}
```

5.2.2 对调试函数进一步封装

上文说到，我们增加的调试语句应能很方便的从最终发行版中去掉，因此我们不能直接调用 printf 或者自定义的 UARTprintf 函数，需要将这些调试函数做一层封装，以便随时从代码中去除这些调试语句。参考方法如下：

```
#ifdef MY_DEBUG
#define MY_DEBUGF(message) do { \
                                {UARTprintf message;} \
                            } while(0)
#else
#define MY_DEBUGF(message)
#endif /* PLC_DEBUG */
```

在我们编码测试期间，定义宏 MY_DEBUG，并使用宏 MY_DEBUGF（注意比前面那个宏多了一个‘F’）输出调试信息。经过预处理后，宏 MY_DEBUGF(message)会被 UARTprintf message 代替，从而实现了调试信息的输出；当正式发布时，只需要将宏 MY_DEBUG 注释掉，经过预处理后，所有 MY_DEBUGF(message)语句都会被空格代替，而从将调试信息从代码中去除掉。

6.编程思想

6.1 编程风格

《计算机程序的构造和解释》一书在开篇写到：程序写出来是给人看的，附带能在机器上运行。

6.1.1 整洁的样式

使用什么样的编码样式一直都颇具争议性的，比如缩进和大括号的位置。因为编码的样式也会影响程序的可读性，面对一个乱放括号、对齐都不一致的源码，我们很难提起阅读它的兴趣。我们总要看别人的程序，如果彼此编码样式相近，读起源码来会觉得比较舒适。但是编码风格的问题是主观的，永远不可能在编码风格上达成统一意见。因此只要你的编码样式整洁、结构清晰就足够了。除此之外，对编码样式再没有其它要求。

提出匈牙利命名法的程序员、前微软首席架构师 Charles Simonyi 说：我觉得代码清单带给人的愉快同整洁的家差不多。你一眼就能分辨出家里是杂乱无章还是整洁如新。这也许意义不大。因为光是房子整洁说明不了什么，它仍可能藏污纳垢！但是第一印象很重要，它至少反映了程序的某些方面。我敢打赌，我在 3 米开外就能看出程序拙劣与否。我也许没法保证它很不错，但如果从 3 米外看起来就很糟，我敢保证这程序写得不用心。如果写得不用心，那它在逻辑上也许就不会优美。

6.1.2 清晰的命名

变量、函数、宏等等都需要命名，清晰的命名是优秀代码的特点之一。命名的要点之一是名称应能清晰的描述这个对象，以至于一个初级程序员也能不费力的读懂你的代码逻辑。我们写的代码主要给谁看是需要思考的：给自己、给编译器还是给别人看？我觉得代码最主要的是给别人看，其次是给自己看。如果没有一个清晰的命名，别人在维护你的程序时很难在整个全貌上看清代码，因为要记住十多个以上的糟糕命名的变量是件非常困难的事；而且一段时间之后你回过头来看自己的代码，很有可能不记得那些糟糕命名的变量是什么意思。

为对象起一个清晰的名字并不是简单的事情。首先能认识到名称的重要性需要有一个过程，这也许跟谭式 C 程序教材被大学广泛使用有关：满书的 a、b、c、x、y、z 变量名是很难在关键的初学阶段给人传达优秀编程思想的；其次如何恰当的为对象命名也很有挑战性，要准确、无歧义、不罗嗦，要对英文有一定水平，所有这些都要满足时，就会变得很困难；此外，命名还需要考虑整体一致性，在同一个项目中要有统一的风格，坚持这种风格也并不容易。

关于如何命名，Charles Simonyi 说：面对一个具备某些属性的结构，不要随随便便地取个名字，然后让所有人去琢磨名字和属性之间有什么关联，你应该把属性本身，用作结构的名字。

6.1.3 恰当的注释

注释向来也是争议之一，不加注释和过多的注释我都是反对的。不加注释的代码显然是很糟糕的，但过多的注释也会妨碍程序的可读性，由于注释可能存在的歧义，有可能会误解程序真实意图，此外，过多的注释会增加程序员不必要的时间。如果你的编码样式整洁、命名又很清晰，那么，你的代码可读性不会差到哪去，而注释的本意就是为了便于理解程序。

这里建议使用良好的编码样式和清晰的命名来减少注释，对模块、函数、变量、数据结构、算法和关键代码做注释，应重视注释的质量而不是数量。如果你需要一大段注释才能说清楚程序做什么，那么你应该注意了：是否是因为程序变量命名不够清晰，或者代码逻辑过于混乱，这个时候你应该考虑的可能就不是注释，而是如何精简这个程序了。

6.2 数据结构

数据结构是程序设计的基础。在设计程序之前，应该先考虑好所需要的数据结构。

前微软首席架构师 Charles Simonyi：编程的第一步是想象。就是要在脑海中对来龙去脉有极为清晰的把握。在这个初始阶段，我会使用纸和铅笔。我只是信手涂鸦，并不写代码。我也许会画些方框或箭头，但基本上只是涂鸦，因为真正的想法在我脑海里。我喜欢想象那些有待维护的结构，那些结构代表着我想编码的真实世界。一旦这个结构考虑得相当严谨和明确，我便开始写代码。我会坐到终端前，或者换在以前的话，就会拿张白纸，开始写代码。这相当容易。我只要把头脑中的想法变换成代码写下来，我知道结果应该是什么样的。大部分代码会水到渠成，不过我维护的那些数据结构才是关键。我会先想好数据结构，并在整个编码过程中将它们牢记于心。

开发过以太网和操作系统 SDS 940 的 Butler Lampson：（程序员）最重要的素质是能够把问题的解决方案组织成容易操控的结构。

开发 CP/M 操作系统的 Gary.A：如果不能确认数据结构是正确的，我是决不会开始编码的。我会先画数据结构，然后花很长时间思考数据结构。在确定数据结构之后我就开始写一些小段的代码，并不断地改善和监测。在编码过程中进行测试可以确保所做的修改是局部的，并且如果有什么问题的话，能够马上发现。

微软创始人比尔·盖茨：编写程序最重要的部分是设计数据结构。接下来重要的部分是分解各种代码块。

编写世界上第一个电子表格软件的 Dan Bricklin：在我看来，写程序最重要的部分是设计数据结构，此外，你还必须知道人机界面会是什么样的。

我们举个例子来说明。在介绍防御性编程的时候，提到公司使用的 LCD 显示屏抗干扰能力一般，为了提高 LCD 的稳定性，需要定期读出 LCD 内部的关键寄存器值，然后跟存在 Flash 中的初始值相比较。需要读出的 LCD 寄存器有十多个，从每个寄存器读出的值也不尽相同，从 1 个到 8 个字节都有可能。如果不考虑数据结构，编写出的程序将会很冗长。

```
1.  void lcd_redu(void)
2.  {
3.      读第一个寄存器值;
4.      if(第一个寄存器值==Flash 存储值)
5.      {
6.          读第二个寄存器值;
7.          if(第二个寄存器值==Flash 存储值)
8.          {
```

```
                ...

                读第十个寄存器值;
                if(第十个寄存器值==Flash 存储值)
                {
                    返回;
                }
                else
                {
                    重新初始化 LCD;
                }
            }
            else
            {
                重新初始化 LCD;
            }
        }
        else
        {
            重新初始化 LCD;
        }
}
```

我们分析这个过程，发现能提取出很多相同的元素，比如每次读 LCD 寄存器都需要该寄存器的命令号，都会经过读寄存器、判断值是否相同、处理异常情况这一过程。所以我们可以提取一些相同的元素，组织成数据结构，用统一的方法去处理这些数据，将数据与处理过程分开来。

我们可以先提取相同的元素，将之组织成数据结构：

```
typedef struct {
    uint8_t   lcd_command;              //LCD 寄存器
    uint8_t   lcd_get_value[8];         //初始化时写入寄存器的值
    uint8_t   lcd_value_num;            //初始化时写入寄存器值的数目
}lcd_redu_list_struct;
```

这里 lcd_command 表示的是 LCD 寄存器命令号；lcd_get_value 是一个数组，表示寄存器要初始化的值，这是因为对于一个 LCD 寄存器，可能要初始化多个字节，这是硬件特性决定的；lcd_value_num 是指一个寄存器要多少个字节的初值，这是因为每一个寄存器的初值数目是不同的，我们用同一个方法处理数据时，是需要这个信息的。

就本例而言，我们将要处理的数据都是事先固定的，所以定义好数据结构后，我们可以将这些数据组织成表格：

```
/*LCD 部分寄存器设置值列表*/
lcd_redu_list_struct const lcd_redu_list_str[]=
```

```
3.  {
4.      {SSD1963_Get_Address_Mode,{0x20}                                    ,1},
/*1*/
5.      {SSD1963_Get_Pll_Mn         ,{0x3b,0x02,0x04}                       ,3}, /*2*/
6.      {SSD1963_Get_Pll_Status   ,{0x04}                                   ,1}, /*3*
7.      {SSD1963_Get_Lcd_Mode       ,{0x24,0x20,0x01,0xdf,0x01,0x0f,0x00}   ,7}, /*4*/
8.      {SSD1963_Get_Hori_Period ,{0x02,0x0c,0x00,0x2a,0x07,0x00,0x00,0x00},8}, /*5*/
9.      {SSD1963_Get_Vert_Period ,{0x01,0x1d,0x00,0x0b,0x09,0x00,0x00}      ,7}, /*6*/
10.     {SSD1963_Get_Power_Mode   ,{0x1c}                                   ,1},
/*7*/
11.     {SSD1963_Get_Display_Mode,{0x03}                                    ,1}, /*8*/
12.     {SSD1963_Get_Gpio_Conf    ,{0x0F,0x01}                              ,2}, /*9*/
13.     {SSD1963_Get_Lshift_Freq ,{0x00,0xb8}                               ,2}, /*10*
14. };
```

至此，我们就可以用一个处理过程来完成数十个 LCD 寄存器的读取、判断和异常处理了：

```
1.  /**
2.  * lcd  显示冗余
3.  *  每隔一段时间调用该程序一次
4.  */
5.  void lcd_redu(void)
6.  {
7.      uint8_t    tmp[8];
8.      uint32_t i,j;
9.      uint32_t lcd_init_flag;
10.
11.     lcd_init_flag =0;
12.     for(i=0;i<sizeof(lcd_redu_list_str)/sizeof(lcd_redu_list_str[0]);i++)
13.     {
14.         LCD_SendCommand(lcd_redu_list_str[i].lcd_command);
15.         uyDelay(10);
16.         for(j=0;j<lcd_redu_list_str[i].lcd_value_num;j++)
17.         {
18.             tmp[j]=LCD_ReadData();
19.             if(tmp[j]!=lcd_redu_list_str[i].lcd_get_value[j])
20.             {
21.                 lcd_init_flag=0x55;
22.                 //一些调试语句，打印出错的具体信息
23.                 goto handle_lcd_init;
24.             }
25.         }
26.     }
27.
```

```
28.     handle_lcd_init:
29.     if(lcd_init_flag==0x55)
30.     {
31.         //重新初始化 LCD
32.         //一些必要的恢复措施
33.     }
34. }
```

通过合理的数据结构，我们可以将数据和处理过程分开，LCD 冗余判断过程可以用很简洁的代码来实现。更重要的是，将数据和处理过程分开更有利于代码的维护。比如，通过实验发现，我们还需要增加一个 LCD 寄存器的值进行判断，这时候只需要将新增加的寄存器信息按照数据结构格式，放到 LCD 寄存器设置值列表中的任意位置即可，不用增加任何处理代码即可实现！这仅仅是数据结构的优势之一，使用数据结构还能简化编程，使复杂过程变的简单，这个只有实际编程后才会有更深的理解。

7.总结和阅读书目

本文介绍了编写优质嵌入式 C 程序涉及的多个方面。每年都有亿万计的 C 程序运行在单片机、ARM7、Cortex-M3 这些微处理器上，但在这些处理器上如何编写优质高效的 C 程序，几乎没有书籍做专门介绍。本文试图在这方面做一些努力。编写优质嵌入式 C 程序需要大量的专业知识，本文虽尽力描述编写嵌入式 C 程序所需要的各种技能，但本文却无力将每一个方面都面面俱到的描述出来，所以本文最后会列举一些阅读书目，这些书大多都是真正大师的经验之谈。站在巨人的肩膀上，可以看的更远。

7.1 关于语言特性

Stephen Prata 著 云巅工作室 译 《C Primer Plus（第五版）中文版》

Andrew Koenig 著 高巍 译 《C 陷阱与缺陷》

Peter Van Der Linden 著 徐波 译 《C 专家编程》

陈正冲 编著 《C 语言深度解剖》

7.2 关于编译器

杜春雷 编著 《ARM 体系结构与编程》

Keil MDK 编译器帮助手册

7.3 关于防御性编程

MISRA-C-:2004 Guidelines for the use of the C language in criticalsystems

Robert C.Seacord 著 徐波 译 《C 安全编码标准》

7.4 关于编程思想

Pete Goodliffe 著 韩江、陈玉 译 《编程匠艺---编写卓越的代码》

Susan Lammers 著 李琳骁、吴咏炜、张菁《编程大师访谈录》