‘安全第一’的 C 语言编程规范

作 者：清华大学 陈萌萌 邵贝贝

编者按： C 语言是开发嵌入式应用的主要工具，然而 C 语言并非是专门为嵌入式系统设计，相当多的嵌入式系统较一般计算机系统对软件安全性有更苛刻的要求。1998 年，MISRA 指出，一些在 C 看来可以接受，却存在安全隐患的地方有 127 处之多。2004 年，MISRA 对 C 的限制增加到 141 条。

嵌入式系统应用工程师借用计算机专家创建的 C 语言，使嵌入式系统应用得以飞速发展，而 MISRAC 是嵌入式系统应用工程师对 C 语言嵌入式应用做出的贡献。如今 MISRA C 已经被越来越多的企业接受，成为用于嵌入式系统的 C 语言标准，特别是对安全性要求极高的嵌入式系统，软件应符合 MISRA 标准。

从本期开始，本刊将分 6 期，与读者共同学习 MISRAC。

第一讲：“‘安全第一’的 C 语言编程规范”，简述 MISRAC 的概况。

第二讲：“跨越数据类型的重重陷阱”，介绍规范的数据定义和操作方式，重点在隐式数据类型转换中的问题。

第三讲：“指针、结构体、联合体的安全规范”，解析如何安全而高效地应用指针、结构体和联合体。

第四讲：“防范表达式的失控”，剖析 MISRAC 中关于表达式、函数声明和定义等的不良使用习惯，最大限度地减小各类潜在错误。

第五讲：“准确的程序流控制”，表述 C 语言中控制表达式和程序流控制的规范做法。

第六讲：“构建安全的编译环境”，讲解与编译器相关的规范编写方式，避免来自编译器的隐患。

C/C++语言无疑是当今嵌入式开发中最为常见的语言。早期的嵌入式程序大都是用汇编语言开发的，但人们很快就意识到汇编语言所带来的问题——难移植、难复用、难维护和可读性极差。很多程序会因为当初开发人员的离开而必须重新编写，许多程序员甚至连他们自己几个月前写成的代码都看不懂。C/C++语言恰恰可以解决这些问题。作为一种相对“低级”的高级语言，C/C++语言能够让嵌入式程序员更自由地控制底层硬件，同时享受高级语言带来的便利。对于 C 语言和 C++语言，很多的程序员会选择 C 语言，而避开庞大复杂的 C++语言。这是很容易理解的——C 语言写成的代码量比 C++语言的更小些，执行效率也更高。

对于程序员来说，能工作的代码并不等于“好”的代码。“好”代码的指标很多，包括易读、易维护、易移植和可靠等。其中，可靠性对嵌入式系统非常重要，尤其是在那些对安全性要求很高的系统中，如飞行器、汽车和工业控制中。这些系统的特点是：只要工作稍有偏差，就有可能造成重大损失或者人员伤亡。一个不容易出错的系统，除了要有很好的硬件设计（如电磁兼容性），还要有很健壮或者说“安全”的程序。

然而，很少有程序员知道什么样的程序是安全的程序。很多程序只是表面上可以干活，还存在着大量的隐患。当然，这其中也有 C 语言自身的原因。因为 C 语言是一门难以掌握的语言，其灵活的编程方式和语法规则对于一个新手来说很可能会成为机关重重的陷阱。同时，C 语言的定义还并不完全，即使是国际通用的 C 语言标准，也还存在着很多未完全定义的地方。要求所有的嵌入式程序员都成为 C 语言专家，避开所有可能带来危险的编程方式，是不现实的。最好的方法是有一个针对安全性的 C 语言编程规范，告诉程序员该如何做。

1 MISRAC 规范

1994 年，在英国成立了一个叫做汽车工业软件可靠性联合会（The Motor Industry Software Reliability Association，以下简称 MISRA）的组织。它是致力于协助汽车厂商开发安全可靠的软件的跨国协会，其成员包括：AB 汽车电子、罗孚汽车、宾利汽车、福特汽车、捷豹汽车、路虎公司、Lotus 公司、MIRA 公司、Ricardo 公司、TRW 汽车电子、利兹大学和福特 VISTEON 汽车系统公司。

经过了四年的研究和准备，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》(Guidelines for the Use of the C Language in Vehicle Based Software)，共有 127 条规则，称为 MISRAC:1998。

C 语言并不乏国际标准。国际标准化组织（International Organization of Standardization,简称 ISO）的“标准 C 语言”经历了从 C90、C96 到 C99 的变动。但是，嵌入式程序员很难将 ISO 标准当作编写安全代码的规范。一是因为标准 C 语言并不是针对代码安全的，也并不是专门为嵌入式应用设计的；二是因为“标准 C 语言”太庞大了，很难操作。MISRAC:1998 规范的产生恰恰弥补了这方面的空白。

随着很多汽车厂商开始接受 MISRAC 编程规范，MISRAC:1998 也成为汽车工业中最为著名的有关安全性的 C 语言规范。2004 年，MISRA 出版了该规范的新版本——MISRAC: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 MISRAC: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 MISRAC: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。

MISRAC:2004 将其 141 条规则分为 21 个类别，每一条规则对应一条编程准则。详细情况如表 1 所列。

表 1MISRAC:2004 规则分类

分类	强制规则	推荐规则	分类	强制规则	推荐规则
开发环境	4	1	表达式	9	4
语言外延	3	1	控制表达式	6	1
注释	5	1	控制流	10	0
字符集	2	0	Switch 语句	5	0
标识符	4	3	函数	9	1
类型	4	1	指针和数组	5	1
常量	1	0	结构体和联合体	4	0
声明和定义	12	0	预处理命令	13	4
初始化	3	0	标准库	12	0
算术类型转换	6	0	运行失败	1	0
指针类型转换	3	2	—	—	—

最初，MISRAC:1998 编程规范的建立是为了增强汽车工业软件的安全性。可能造成汽车事故的原因有很多，如图 1 所示，设计和制造时埋下的隐患约占总数的 15%，其中也包括软件的设计和制造。MISRAC:1998 就是为了减小这部分隐患而制定的。

MISRAC 编程规范的推出迎合了很多汽车厂商的需要，因为一旦厂商在程序设计上出现了问题，用来补救的费用将相当可观。1999 年 7 月 22 日，通用汽车公司（General Motors）就曾经因为其软件设计上的一个问题，被迫召回 350 万辆已经出厂的汽车，损失之大可想而知。

MISRAC 规范不仅在汽车工业开始普及，也同时影响到了嵌入式开发的其他方向。嵌入式实时操作系统 μC/OSII 的 2.52 版本虽然已经于 2000 年通过了美国航空管理局（FAA）的安全认证，但 2003 年作者就根据 MISRAC:1998 规范又对源码做了相应的修改，如将

```
if ((pevent->OSEventTbl[y] &= ~bitx) == 0) {
    /*… */
}
```

的写法，改写成

```
pevent->OSEventTbl[y] &= ~bitx;
if (pevent->OSEventTbl[y] == 0) {
/*… */
}
```

发布了 2.62 的新版本，并宣称其源代码 99％符合 MISRAC:1998 规范。

一个程序能够符合 MISRAC 编程规范，不仅需要程序员按照规范编程，编译器也需要对所编译的代码进行规则检查。现在，很多编译器开发商都对 MISRAC 规范有了支持，比如 IAR 的编译器就提供了对

MISRAC:1998 规范 127 条规则的检查功能。

2 MISRAC 对安全性的理解

MISRAC:2004 的专家们大都来自于软件工业或者汽车工业的知名公司，规范的制定不仅仅像过去一样局限于汽车工业的 C 语言编程，同时还涵盖了其他高安全性系统。

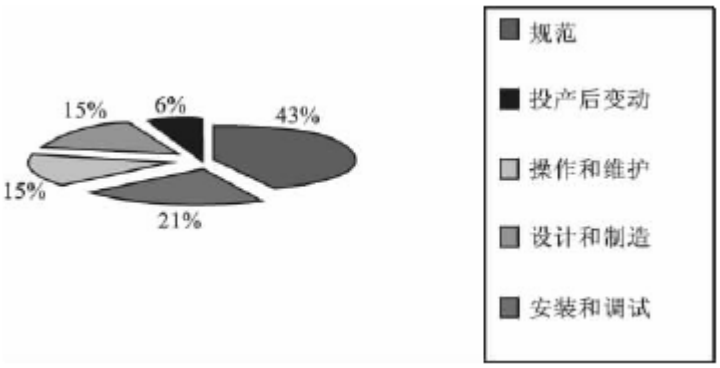

图 1 汽车事故原因分布图

MISRAC: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)。

程序员的失误是司空见惯的。程序员是人，难免会犯错误。很多由程序员犯下的错误可以被编译器及时地纠正（如键入错误的变量名等），但也有很多会逃过编译器的检查。相信任何一个程序员都曾经犯过将“= =”误写成“=”的错误，编译器可能不会认为

```
if(x=y)
```

是一个程序员的失误。

再举个例子，大家都知道++运算符。假如有下面的指令：

```
i=3;
printf(“%d”,++i);
```

输出应该是多少？如果是:

```
printf(“%d”,i++);
```

呢？如果改成-i++呢？i+++i 呢？i+++++i 呢？绝大多数程序员恐怕已经糊涂了。在 MISRAC:2004 中，会明确指出++或--运算符不得和其他运算符混合使用。

C 语言非常灵活，它给了程序员非常大的自由。但事情有好有坏，自由越大，犯错误的机会也就越多。

如果说有些错误是程序员无心之失的话，那么因为程序员对 C 语言本身或是编译器特性的误解而造成的错误就是“明”知故犯了。C 语言有一些概念很难掌握，非常容易造成误解，如表达式的计算。请看下面这条语句：

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

很多程序员认为执行了这条指令后，i 变量的值就会自动加 1。但真正的情况如何呢？MISRA 中有一条规则：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=i++之后，i 的值会发生变化。

另外，不同编译器对同一语句的处理可能是不一样的。例如整型变量的长度，不同编译器的规定就不同。这就要求程序员不仅要清楚 C 语言本身的特性，还要了解所用的编译器，难度很大。

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。例如：

```
#define NULL 0
    ……
    char* p;
    p=NULL;
    printf(“Location of 0 is %d\n”, *p);
```

语法上没有任何问题，但在某些系统上却可能运行出错。

C 语言可以产生非常紧凑、高效的代码，一个原因就是 C 语言提供的运行错误检查功能很少，虽然运行效率得以提高，但也降低了系统的安全性。

有句话说得好，“正确的观念重于一切”。MISRAC 规范对于嵌入式程序员来讲，一个很重要的意义就是提供给他们一些建议，让他们逐渐树立一些好的编程习惯和编程思路，慢慢摒弃那些可能存在风险的编程行为，编写出更为安全、健壮的代码。比如，很多嵌入式程序员都会忽略注释的重要性，但这样的做法会降低程序的可读性，也会给将来的维护和移植带来风险。嵌入式程序员经常要接触到各种的编译器，而很多 C 程序在不同编译器下的处理是不一样的。MISRAC:2004 有一条强制规则，要求程序员把所有和编译器特性相关的 C 语言行为记录下来。这样在程序员做移植工作时，风险就降低了。

3 MISRAC 的负面效应

程序员可能会担心采用 MISRAC:2004 规范会对他们的程序有负面影响，比如可能会影响代码量、执行效率和程序可读性等。应该说，这种担心不无道理。纵观 141 条 MISRAC:2004 编程规范，大多数的规则并不会对程序的代码量、执行效率和可读性造成什么大的影响；一部分规则可能会以增加存储器的占用空间为代价来增加执行效率，或者增加代码的可读性；但是，也确实存在着一些规则可能会降低程序的执行效率。

一个典型的例子就是关于联合体的使用。MISRAC:2004 有一条规则明确指出：不得使用联合体。这是因为，在联合体的存储方式（如位填充、对齐方式、位顺序等）上，各种编译器的处理可能不同。比如，经常会有程序员这样做：一边将采集得到的数据按照某种类型存入一个联合体，而同时又采用另外一种数据类型将该数据读出。如下面这段程序：

```
typedef union{
    uint32_t word;
    uint8_t bytes[4];
}word_msg_t;
unit32_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);
}
```

原理上，这种联合体很像是一个硬件上的双口 RAM 存储器。但程序员必须清楚，这种做法是有风险的。MISRAC:2004 推荐用下面这种方法来做：

```
 uint32_t read_word_big_endian (void) {
    uint32_t word;
    word=((unit32_t)read_byte())<<24;
    word=word|(((unit32_t)read_byte())<<16);
```

```
    word=word|(((unit32_t)read_byte())<<8);
    word=word| ((unit32_t)read_byte());
    return(word);
}
```

先不论为什么这样做会更安全，只谈执行效率，这种采用二进制数移位的方法远远不如使用联合体。到底是使用更安全的做法，还是采用效率更高的做法，需要程序员权衡。对于一些要求执行效率很高的系统，使用联合体仍然是可以接受的方法。当然，这是建立在程序员充分了解所用编译器的基础上的，而且程序员必须对这种做法配有相应的注释。

4 发展中的 MISRAC

MISRAC 并非完美，它自身的发展也印证了这一点。MISRAC:2004 就去掉了 MISRAC:1998 中的 15 条规则。今后的发展，MISRAC 仍然要解决很多问题。比如，MISRAC:2004 是基于 C90 标准的，但最新的国际 C 标准是 C99，而 C99 中没有确切定义的 C 语言特性几乎比 C90 多了一倍，MISRAC 如何适应新的标准还需要进一步探讨。

另外，C++在嵌入式应用中也越来越受到重视，MISRA 正在着手制定 MISRAC++编程规范。读者可以通过访问网站 http://www.misra.org.uk 了解 MISRAC 的发展动向。

5 对 MISRAC 的思考

嵌入式系统并不算是一个独立的学科，但作为一个发展中的行业，它确实需要有一些自己的创新之处。嵌入式工程师们不应仅仅局限于从计算机专家那里学习相关理论知识，并运用于自己的项目，还应该共同努力去完善自己行业的标准和规范，为嵌入式系统的发展做出贡献。MISRAC 编程规范就是一个很好的典范。它始于汽车工程师和软件工程师经验的总结，然后逐渐发展成为一种对整个嵌入式行业都有指导意义的规范。对于推动整个嵌入式行业的正规化发展，MISRAC 无疑有着重要意义。

从另一个角度讲，MISRAC 规范也可以看成是嵌入式工程师对软件业的一种完善。嵌入式工程师虽然不是计算机专家，但却对嵌入式应用有着最深刻的了解，将自己在嵌入式应用中的经验和体会贡献给其他行业，也是他们应该肩负的责任。

参考文献

1 MISRAC:2004, Guidelines for the use of the C language in critical systems. The Motor Industry Software Reliability Association, 2004

2 Harbison III. Samuel P, Steele Jr. Guy L. C 语言参考手册. 邱仲潘，等译. 第 5 版. 北京：机械工业出版社，2003

3 Kernighan. Brian W, Ritchie. Dennis M. C 程序设计语言. 徐宝文，等译. 第 2 版. 北京：机械工业出版社，2001

4 Koenig Andrew. C 陷阱与缺陷. 高巍译. 北京：人民邮电出版社，2002

5 McCall Gavin. Introduction to MISRAC:2004, Visteon UK, http://www.MISRAC2.com/

6 Hennell Mike. MISRA CIts role in the bigger picture of critical software development, LDRA. http://www.MISRAC2.com/

7 Hatton Les. The MISRA C Compliance Suite—The next step, Oakwood Computing. http://www.MISRAC2.com/

8 Montgomery Steve. The role of MISRA C in developing automotive software, Ricardo Tarragon. http://www.MISRAC2.com/