嵌入式C 标准研究报告

（一）

---MISRA C标准工程师笔记

VER: 1.01

PAGE: 40

编辑：肖燕、张墨

关键词：MISRA C 可靠性 安全 编程规范

嵌入式软件国际标准研究
组建专家组

中国单片机公共实验室
2007 年 5 月

前 言

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而言对软件安全性（可靠性）有更苛刻的要求，所以因此会带来更多的安全隐患。

丰田汽车已经表示要对 2005 年 10、2003 年 8 月至 2004 年 11 月生产的约 16 万辆混合动力汽车“普锐斯”进行无偿修理。据称，主要是发动机的 ECU 程序出了问题，行驶中发动机会突然停止。此外宝马公司 2003 年 7 月也因发动机 ECU 的软件问题而提出召回缺陷汽车。1999 年 7 月 22 日，通用汽车公司（General Motors）也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。同样，在电梯和医疗器械产品上也出现过类似的严重问题。

由此可以看出软件质量问题已经越来越深刻的影响到了产品的质量，甚至有些时候是致命的，在航空航天等领域更是如此。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‘入门容易，得道难’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。MISRA C 因此应运而生。

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，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 MISRA C:1998。目前 MISRA C:2004 版已有 141 条规则，21 个类别，每一条规则对应一条编程准则。

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。在未来，MISRA C 也趋向于成为国际性的嵌入式 C 语言开发标准规范。未来全部的嵌入式产品可能必须符合 MISRA C 标准。所以要提高产品的竞争力，避免国际

标准化推广使中国嵌入式系统开发企业处于被动，中国计算机学会嵌入式系统专业委员会建议联合中国汽车工程学会、中国系统工程学会和 ISO 国际标准化组织相关专家组在中国开展**嵌入式系统安全规范开发研究工作并组织中国专家工作组**。通过引进、消化、吸收、再创新为中国的嵌入式系统企业学习 MISRA C 标准提供支持，共同分享先进的国际标准化信息和技术文档以提升中国嵌入式系统工程师软件水平，与国际标准化组织的最新成果保持同步。

目前已有众多国内知名公司和权威机构加入到了嵌入式系统安全规范开发研究专家组，如：清华大学汽车系、北京英贝多公司、北京集成电路设计园等。共同致力于提升中国嵌入式研究能力。

以下的文章是一些工程师的 MISRA C 的学习笔记，推荐给各位参考。笔记中的某些观点并不全面，表达上也不够准确与国际标准的最新进展是有差距的，但对了解 MISRA C 有一定的参考价值。从中国发展嵌入式系统战略的角度看，我们需要组织国内专家进行更深入系统的对 MISRA C 和嵌入式系统 C 语言标准进行研究和持续跟踪，以有效提升中国嵌入式工程师的水平。在 SOC 开发方面和单片机应用方面，对嵌入式 C 语言标准化和规范化有着强烈的需求，这也是保证我国嵌入式系统工业发展的基础环节。希望有志之士能够加入我们的专家组。

天下大事必作于细。（中国老子）　魔鬼长存于细节之中。（西方谚语）

如想了解更多 MISRA C 和专家组的信息请与中国单片机公共实验室联系。

TEL：010-82357581

Email：support@bol-system.com

Web: http://www.bol-system.com

合作平台和技术资源：

中国汽车工程学会汽车电子技术分会

中国计算机学会 嵌入式系统专业委员会

中国系统工程学会信息系统工程专业委员会

国际标准化组织嵌入式软件标准化专家组 ISO/WG1 /TF1

2007 年 5 月 25 日

学习 MISRA C 之一:

“安全第一”的 C 语言编程规范

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。

第一讲：“‘安全第一’的 C 语言编程规范”，简述 MISRAC 的概况。

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 1 MISRAC: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。

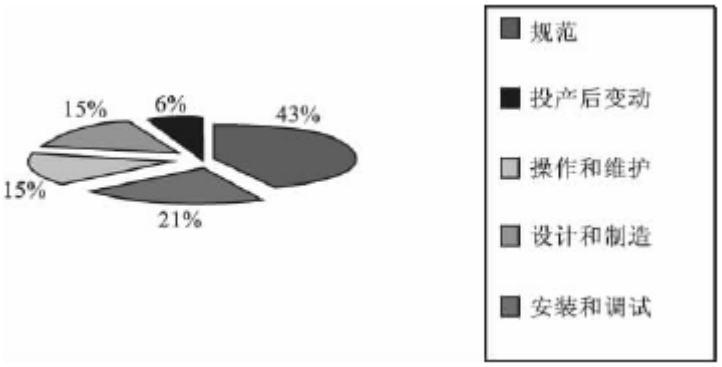

图 1 汽车事故原因分布图

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/**

学习 MISRA C 之二:

跨越数据类型的重重陷阱

数据类型是编程语言中最基本的构成元素，但却是最易被忽略的一环，程序员愿意把几乎100％的精力都花在算法研究、程序流控制等大环节上，却很少在数据类型问题上反复斟酌。

细节决定成败，一个螺丝钉的失误可能导致一个飞行器的毁灭，一个数据类型的错误同样可以让庞大的软件系统崩溃。

MISRA—c 中关于数据类型的规则主要分为两个方面。一是数据类型相关的编程风格；二是不同数据类型之间的转换，后者是重点。这里介绍 MISRA_C 关于数据类型的部分规则，更多的规则请参考《MISRA-C：2OO4)》一书。

下文中凡是未加特殊说明的都是强制(required)规则．个别推荐(advisory)规则加了“推荐”标识。

在展开论述之前，先看两个问题，读者可以带着疑问阅读完本章内容。

问题 1：执行以下程序，result_8 的值是多少?

```
uint8_t port = 0x5a ;
uint8_t result_8 ;
result_8 = (～port) > > 4 ;
／*注：uint8_t 表示 8 位无符号整型* /
```

问题 2：执行以下程序，d 的值是多少?

```
uint16_t a = 10 ;
uint16_t b = 65531 ;
uint32_t c = 0 ;
uint32_t d ;
d = a + b + c ;
／*注：uintl6_t 表示 16 位无符号整型，uint32_t 表示 32 位无符号整型* /
```

1 数据类型相关的编程风格

规则 6.3(推荐)：必须用 typedef 显式标识出各数据
类型的长度和符号特性，避免直接使用标准数据类型。
例如，一个 32 位的整数系统，可定义如下：

```
typedef char chat_t;
typedef sigrled char int8_t;
```

typedef signed short intl6_t；
typedef signed int int32_t；
typedef signed long int64_t；
typedef unsitgned chat uint8_t；
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;
typedef unsigned 1ong uint64_t；

之所以用 intl6_t 和 uint32_t 等代替 signed short 和 unsigned int 等标准数据类型标识符，是由于不同的编译器对标准数 据类型的长度定义是不一样的。比如说一个 16 位系统，很可能就把 short 和 int 都定义成 16 位，long 定义成 32 位，这与上文 32 位系统中标准数据 类型的长度就不一致。用 intl6_t 和 uint_32 等标识符来定义变量，一方面增加了程序的可读性，使得程序员本人或其他读者都能对程序中数据的具体 信息胸有成竹；另一方面也有助于程序在不同系统之间的移植，节省开发时间，减少隐患。规则 7 1：不得使用八进制常数(O 除外)或八进制转义符。
思考如下数组：
code[1]=109；
code[2]=100；
code[3]=O52
code[4]=O71；
／*注：八进制常数须在最高位加 O*／

code[3]的实际值是42(十进制)，code[4]的实际值是57(十进制)；但估计很多读者会把code[3]认成是 52(十进制)，code[4]认成是 7l(十进制)。

八进制数在 C 程序中使用的频率远小于十进制数和十六进制数，为了保证程序的可读性和安全性，程序员不允许使用八进制数以及八进制转义符。

2 数据类型转换

如果程序员对数据类型的转换有很清晰的认识，并且在必要的地方做了正确的显式强制转换，那程序是安全的。但有时由于程序员的疏忽，或者是过于相信编译器的 “智慧”程度，导致表达式中有很多隐式转换(即没有显式地强制转换)，而这些隐式数据类型转换很可能就构成致命的漏洞。MISRA—C 中数据类型转换规则 的着眼点，即是避免有漏洞的隐式数据转换。

在介绍 MISRA—C 关于数据类型转换的部分规则之前，先介绍整型操作数的“平衡(balance)”原则。所谓整型操作数“平衡”原则，即对于隐式表达式，编译器会按照既定规则对操作数进行位数扩充，其中 int 和 unsiglled int 在整型表达式“平衡”过程中占重要地位。

下面分析一个简单的隐式整型表达式 c=a+b(假设 a 的存储位数不大于 b 的存储位数)，编译器是这样来处理这个表达式的：

如果 b 是短整型(即位数少于 int，比如 char、short 等)或者整型(int 或 unsigned int)，那 a 也是短整型或者整型，执行“+”运 算之前，a 和 b 都将被扩充为整型(int 或者 unsigned int)，然后相加的结果赋给 c(如果 c 不是 int 或者 unsigned int 类型，则 这个赋值操作也会包含隐式的扩充或截断操作)。

如果 b 是长整型(存储位数多于 int)，则 a 会被扩充为与 b 相当的长整型，再执行“+”运算，所得结果赋给 c(可能包含隐式的扩充或截断操作)。

绝大部分的操作符用于整型运算的时候，都遵循上述“平衡”原则，比如：算术操作符、位操作符和关系运算符。

但逻辑操作符不遵循上述“平衡”原则。此外左移(<<)和右移(>>)运算符也不遵循“平衡”原则，只和移位操作符左边的整型操 作数相关。假设一个 8 位的短整型变量值为 Oxf5(十六进制)，则右移 4 位所得结果是 O xof(十六进制)。

明确了上述背 景后，下面来关注本文一开始提出的“问题 1”(代码参见前文)。绝大部分拥有嵌人式 C 程序开发经验的人都明白这段代码的原意是将 port 的值取反后右移 4 位赋值给 result_8(在用 I / O 口控制共阳的 LED 时经常这么做)，程序员期望的结果显然是 resuIt_8=0xof。然而，由于整型的“平衡” 原则，在 16 位编译器中，～port 的值是 Oxffa5；在 32 位编译器中，～pott 的值是 Oxfffffa5。无论哪种情况，最后结果(右移 4 位后 赋值给 result_8 的时候有一个截断操作)都是 resuIt_8=Oxfa，而非程序员预期的 result_8=OxOf。

倘若将最后一行代码改成 result 一 8=((uin8_t)(～port))>>4，则 result_8 可取得预期的值。

针对以上情况，MISRA-c 提出了相应规则。

规则 10.5：如果位操作符～和移位操作符<<(或>>)联合作用于 unsigned char 或者 unsigned short 类型的操作数时，中间运算步骤的结果必须立刻显式强制转换为预期的短整型数据类型。

为了加深对“平衡”原则的理解，再来分析一下“问题 2”。

如果用一个 32 位的编译器来编译这段程序，最终结果是 d=6554l，程序员“幸运地”得到了预期的结果。如果是 16 位的编译器，得到的结果却是 d=5。

由于“+”运算是左结合的，所以 d=a+b+c 等效于 d=(a+b)+c，即先执行 a+b，所得的和再与 c 相加．最后结果赋值给 d。问题就出在 a+b 这个 中间步骤中。由于 a 和 b 都是 16 位整型(注意编译器也是 16 位的)，故而 a+b 的结果也是 16 位整型，则 a+b 的值是 Ox0005(有溢出)；再扩充为 32 位整型 Ox00000005 和 c 相加赋值给 d，d=5，这并非程序员预期的结果。

所以，在 16 位编译器中，问题 2 的那段 代码很可能导致严重错误。当然，如果程序员用()指定了运算优先级的话，即最后一行代码写成 d=a+(b+c)，也可以避免上述溢出错误，然而，这终究不 是治本的办法。只有明确每一个操作数的实际数据类型，才能保障代码的安全性。

MISRA-C 中对于表达式中存在隐式数据类型转换的情况作了严格的限制。

规则 10.1：以下情况下，整型表达式中不允许出现隐式数据类型转换。
①整型操作数不是被扩充为更多位数的同符号整数；
②表达式是复杂表达式；
③表达式不是常数表达式，且是函数的参数；
④表达式不是常数表达式，且是函数的返回表达式。。

规则 10.2：以下情况下，浮点数表达式中不允许出现隐式数据类型转换。
①浮点型操作数不是被扩充为更多位数的同符号浮点数；
②表达式是复杂表达式；
③表达式是函数的参数；
④表达式是函数的返回表达式。

整型表达式规则和浮点数表达式规则基本类似，只是浮点数表达式规则更为苛刻一些，对浮点型的常数也作了严格的限定。

这两条规则中，出现了“复杂表达式”的概念。请注意，MISRA—C 中“复杂表达式”的概念和其他介绍 C 编程规范书籍中“复杂表达式”的概念是不一样的。 在 MISRA-C 中，非“复杂表达式”基本只限制在常数表达式或者函数的返回值。为了明确上述规则中关于“复杂表达式”和“返回表达式”的概念，此处举一 例子。定义一个函数 uintl6_t foo(void)，函数体如下：

```
uintl6_t foo(void){
return(a+b+c);
```

函数体中最后一句 return(a+b+c)中的 a+b+c 是返回表达式。倘若在 C 程序的其他地方有 a=foo()这样的语句，则用的是 foo()函数的 返回值。在 MISRA-c 中，的资源，完成了采用 USB 接口技术的热敏打印机的开发，并对打印头作了充分的保护。通过采用相应的算法实现这个赋值表达式不 是“复杂表达式”。

至于表达式作为函数参数等情况，碍于篇幅的原因，此处就不再详细展开了。

权衡 一下利弊，在涉及到数据类型转换的时候，与其花很大力气去区分一个隐式表达式是否在 MISRA—C 规则的“黑名单”中，还不如用强制转换符显式地标识出每 个操作数的实际数据类型，这是最为稳妥的方法。总而言之，MISRA—C 关于数据类型转换规则的中心意思，是要求程序员明确任意一个操作数的实际数据类 型。

3 小结

作为一名优秀程序员，第一步就是以严谨的态度对待程序中的每一个数据，明白任何一个数据操作的关键，从而能写出最清晰易懂而又安全的代码。MISRA—C 关于数据类型的规则可保障程序员在迈出这一步的时候不会摔倒。

学习 MISRA C 之三:

指针结构体联合体的安全规范

指针赋予了 C 编程最大的灵活性;结构体使得 C 程序整齐而紧凑;联合体在某些要求注重效率的场合有精彩的表现.这三个要素是 C 语言的精华.
然而精华并不意味着完美。C 语言在赋予程序员足够的灵活性的同时，也给了程序员许多犯错的机会。所以，有必要关注指针，结构体和联合体的实现细节，从而保障程序的安全性。

在此，第一部分介绍 MISRA-C:2004 中与指针相关的部分规则，第二部分讲解结构体和联合体的操作
规范。下文凡是未加特别说明的都是 强制(required)规则，个别推荐(advisory)规则则加了“推荐”的提示.

指针的安全规范

MISRA-C:2004 关于指针的规范主要分为三个部分：指针的类型转换规则,指针运算的规则和指针的有效性规则。

指针的类型转换

指针类型转换是个高风险的操作，所以应该尽量避免进行这个操作。MISRA-C 对其中可能造成严重错误的情况做了严格的限定，选择其中两条做简要分析。

规则 11.4(推荐) 指向不同数据类型的指针之间不能相互转化。
考察下面的程序

```
uint8_t * p1;
uint32_t * p2;
p2 = (uint32_t *)p1;
/* 注: uint8_t 表示 8 位无符号整型，uint32_t 表示 32 位无符号整型 */
```

程序员希望将从 p1 单元开始的 4 个字节组成一个 32 位的整型来参与运算.

如果 CPU 允许各种数据对象存放在任意的存储单元，则以上转换没有问题。但某些 CPU 对某些(种)数据
类型加强了对齐限制，要求这些数据对象占用一定的地址空间，比如某些字节寻址的 CPU 会要求 32 位
（4 字节)整型存放在 4 的整数倍地址上(也就是所谓的 address alighment 地址对齐).在这个前提下
,思考程序中的指针转化，假设 p1 一开始指向的是 0x0003 单元(对 uint8_t 型的整数没有对齐要求),则
执行最后一行强制转换后，p2 到底指向哪个单元就无法预料了.

规则 11.5 指针转换过程中不允许丢失指针的 const volatile 属性

按照如下定义指针

```
uint16_t x;
uint16_t * const cpi = &x;   /* const 指针 */
unit16_t * const *pcpi ;     /* 指向 const 指针的指针 */
const uint16_t * *ppci ;     /* 指向 const 整数指针的指针 */
uint16_t * * ppi;
const uint16_t * pci;        /* 指向 const 整型的指针 */
volatile uint16_t * pvi;     /* 指向 volatile 整型的指针 */
unit16_t  *pi;
```

则一下指针转换是允许的

```
pi = cpi;
```

以下指针转换是不允许的

```
pi = (uint16_t *) pci;
pi = (uint16_t *) pvi;
ppi = (uint16_t **)pcpi;
ppi = (uint16_t **)ppci;
```

以上非法指针类型转换将会丢失 const 或者 volatile 类型。丢失 const 属性将有可能导致在对只读内容进行写操作，编译器不会发出警告，编译器将对不具有 volatitle 属性的变量做优化;丢失 volatile 属性,编译器的优化可能导致程序员预先设计的硬件时序操作失效，这样的错误很难发现。关于 const 和 volatile 关键字的详细左右，读者可参考 ISO C 获取更多信息.

指针的运算

ISO C 标准中，对指向数组成员的指针运算（包括算术运算，比较等）作了规范定义，除此以外的指针运算属于未定义(undifined）范围,具体实现有赖于具体编译器件，其安全性无法得到保障，

MISRA-C 中对指针运算的合法范围作了如下限定.

规则 17.1 只有指向数组的指针才允许进行算术运算。

(注:其实此处的算术运算子仅仅有限定于指针加减某个整数，比如 ppoint = point -5, ppoint++等)

规则 17.2 只有指向同一个数组的两个指针才允许相减.

(两个指针 可指向 同一数组的不同成员)

规则 17.3 只有指向同一个数组的两个指针才允许用>, >=, < , <=等关系运算符进行比较。

为了尽可能减少直接进行指针运算带来的隐患，尤其是程序动态运行时可能发生的数组越界等问题，MISRA-C 对指针运算作了更为严格的规定。

规则 17.4 只允许用数组索引做指针运算。

按照如下方式定义数组和指针:
uint8_t a[10];
unit8_t *p;
p = a;
则*(p+5) = 0 是不允许的，而 p[5] = 0 则是允许的，尽管就这段程序而言，二者等价。

以下给出一段程序，读者可参照相应程序行的注释，细细品位上述规则的含义。

```
void my_fun(uint * _t * p1, uint8_t p2[])
{
    uint8_t index = 0;
      uint8_t *p3;
    uint8_t *p4;
    *p1 = 0;
    p1 ++;          /*  不允许, p1 不是指向数组的指针  */
      p1 = p1 +5;     /*  不允许, p1 不是指向数组的指针  */
    p1[5] = 0;      /*  不允许, p1 不是指向数组的指针  */
      p3 = &p1[5];
      p2[0] = 0;
    index ++;
    index = index + 5;
    p2[index] = 0;     /*允许*/
    *(p2+index) = 0;   /*  不允许  */
    p4 = &p2[5];       /*允许*/
}
```

1.3 指针的有效性

下面介绍 MISRA-C:2004 中关于指针有效性的规则

规则 17.6 动态分配对象的地址栏不允许在本对象消亡后传给另外一个对象

这条规则的实际意义上是 不允许将栈对象的地址 传给外部作用域的对象

请看下面这段程序:

```
#include "stdio.h"
```

```
char * getm(void)
{
    char p[] = "hello world";
    return p;
}

int main()
{
    char *str = NULL;
    str = getm();
    printf(str);
}
```

程序员希望最后的输出结果是"hello world"这个字符串，然而实际运行时，却出现乱码(具体内容依赖于编译运行环境).

简单分析以下，由于 char p[] = "hello,world"这条语句是在栈中分配空间存储"hello world"这个字符串，当函数 getm()返回的时候，已分配的空间将会被释放（但内容并不会被销毁），而 printf(str)涉及系统调用，有数据压栈，会修改从当前分配给数组 p[]存储空间的内容，导致程序无法得到预期的结果.

倘若将 getm()函数体中的 char p[] = "hello world"程序行改成 char *q = "hello world",则执行 main()的时候就可以正确输出"hello world".这事因为这样的方式，q 指向的是静态数据区，而非栈中的某个单元(也就是说 hello world 分配到 bss 区中了而不是栈中)

所以，数组名是指针不假，但在实现细节上还是有很大的差异，程序员在使用指针的时候必须慎之又慎

结构体 联合体的安全规范

规则 18.4 不允许使用联合体

这事一个不太近情理的规定，在具体阐述为何 MISRA-C:2004 如此痛恨联合体之前，首先需要明确与联合体相关的细节:

1)联合体的末尾有多少个填充单元
2）联合体的各个成员如何对齐
3）多字节的数据类型高低字节如何排放顺序
4）如果包含位字段 (bit-field)，各位如何排放

针对细节 3 举个例子
程序段 2.1

```
typedef union{
    uint32_t word;
```

```
    uint8_t bytes[4];
}word_msg_t;

uint32_t read_msg(void)
{
    word_msg_t tmp;
/* tmp.byte[0]  对应于 tmp.word 的高 8 位
   tmp.byte[1]对应于 tmp.word 的次高 8 位,依此类推  */
    tmp.bytes[0]   = read_byte();
    tmp.bytes[1]   = read_byte();
    tmp.bytes[2]   = read_byte();
    tmp.bytes[3]   = read_byte();
    return (tmp.word);
}
```

以上代码在各种通信协议中使用的频率很高，接收端接收到的数据一般都以字节为单位存放，主控程序需要根据相应的协议将接受到的多个字节进行组合。为了实现相同的功能,MISRA-C:2004 推荐恶劣 read_msg()函数的另外一种写法.

程序段 2.2

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

无论从程序的可读性还是从执行效率来讲，程序段 2.1 都要优于程序段 2.2。然而，程序段 2.1 在 Intel 80x86/Pentium 体系(little-endian,存储多字节整数的时候低位字节存放在低地址)CPU 和在 Motorola 68K 体系(big-endian)中的执行结果完全不一样。假设 read_byte()函数返回的数据依次是 0x01,0x02,0x03,0x04,则在 Intel 体系中，程序段 2.1 read_msg 返回的值是 0x4321,在 Motorola 体系中，返回的则是 0x1234.　　无论在 Intel 体系还是在 Motorola 体系中,程序段 2.2 中 read_msg 的返回值都是 0x1234

以上是联合体中多字节整型字节排放顺序不定导致漏洞的一个例子。倘若不明确联合体末尾填充的细节，或者不清楚联合体成员的对齐方式，或者不注意联合体中位字段成员的位排列次序，都有可能导致错误。作为将安全性放在第一位的 C 标准，MISRA_C 禁止使用联合体并非不可理喻

然而，联合体毕竟是 C 语言的一个重要元素，所以 MISRA_C 主张禁止使用联合体的同时，也为效率和资源要求比较苛刻的情况开了一扇们，程序员在明确联合体各个实现细节的前提下，在万不得已的时候，仍可以谨慎使用联合体，在不同体系的 CPU 间移植程序时也要注意做相应的修改.此外，MISRA-C:2004 中也对结构体和联合体的编程风格作了限定

规则 18.1 所有结构体和联合体的定义必须保证完整性

由于涉及 ISO C 中类型定义完整性等概念，碍于篇幅的原因，此处不再赘述，读者可以参阅 MISRA-C:2004 一书，和 ISO C 标准以了解更多信息，完善自己的编程风格

小结

总而言之，对于 C 程序总最灵活的指针，结构体和联合体，程序员不仅仅要关注其定义和操作的一般方法,更要注意实现细节。由于指针 联合体的功能性错误一般都可以逃过编译器的检查，所以稍有疏忽，就可能导致程序在运行的时候出现严重错误，程序员必须以严禁甚至苛刻的态度对待指针，结构体和联合体。

学习 MISRA C 之四:

防范表达式的失控

在 C 语言中,表达式是最重要的组成部分之一,几乎所有的代码都由表达式构成。表达式的使用如此广泛,读者也许会产生这样的疑问,像+ 、- 、3 、/ 、& & 这样简单的运算也会出现问题吗? 程序员在编写表达式时,往往带有一些不良的习惯。即使是编写很简单的表达式,这些不良习惯也可能造成隐患,这个小小的隐患甚至可能引起整个系统的崩溃。实际上,在程序调试过程中,表达式中存在的大部分隐患皆来源于程序员的主观臆测,即认为表达式应该是按自己认为的方式执行,但结果可能完全相反。这是因为程序设计语言或编译器的某些内在机制并不如我们所想的那样。所有的编译器都遵从这一假定:程序员都是“神”,他们既了解编程语言的各种特性,也了解编译器本身一些鲜为人知的处理原则。当然, 程序员不是“神”。因此,程序员在编写程序的过程中需要小心地避免编译器“设置”的各种陷阱,而问题是有些时候很难预测下一步是否会踏上一个陷阱。

MISRA C 规则中包含了大量关于表达式书写的规范,最大程度地防范上述可能发生的错误,告诉程序员如何编写规范的 C 语言表达式。
本文将首先深入剖析在编译器内部表达式的解析方式,然后罗列和分析书写表达式过程中常见的不规范写法,以帮助读者避免各种不良的编程习惯。文中凡是未加特殊说明的都是强制(required) 规则,个别推荐(advisory)规则加了“推荐”标示。

1 表达式的求值顺序

首先,分析下面两段代码。

问题 1 :执行以下程序,从串口依次输入 2 和 4 ,变量 result 将等于多少?

```
uint8_t result ;
result = uart_GetChar () $ uart_GetChar () ;

/ * 这里,uint8_t 表示 8 位无符号整数类型“, uint8_t uart_GetChar () ”是从串口接收一个 ASCII 字符的函数。*/
```

问题 2 :执行以下程序,变量 result 将等于多少?

```
uint8_t result ;
uint8_t temp = 2 ;
result = temp + + + - - temp ;
```

也许读者会不假思索地写出结果:
第一题的答案是“ - 2”,即 result = 0x32 - 0x34 = 0xFE ;
第二题的答案是“4”,即 result = (temp + +) + (- - temp) = 2 + 2 = 4 。

推理过程看起来似乎是正确的,可是,程序的运算结果是这样吗? 为了弄清楚这个问题,看一下 C 语言中有关运算符的规定。

表 1 给出了 C 语言中运算符和结合律的对应关系。
第一行中,++ 和-- 为后缀自增运算符和后缀自减运算符;第二行中,++ 和-- 为前缀自增运算符和前缀自减运算符,+ 、- 、3 、& 分别对应一元正运算符、一元负运算符、间接取值运算符和取地址运算符;其他的+ 、- 、3 、& 分别对应二元加运算符、二元减运算符、二元乘运算符和位与运算符。

表 1 C 语言运算符列表(按优先级从高到低)

优先级	运算符	结合律	类型
从高到低排列	( ) [ ] - > . + + (后缀) -- (后缀)	从左至右	一元运算符
	! ~ + + (前缀) - - (前缀) sizeof (type) (强制类型转换) + - 3 &	从右至左	一元运算符
	* / %	从左至右	二元运算符
	+ -	从左至右	
	< < > >	从左至右	
	< < = > > =	从左至右	
	= = ! =	从左至右	
	&	从左至右	
	^	从左至右	
	|	从左至右	
	& &	从左至右	
	| |	从左至右	
	?:	从右至左	三元运算符
	= + = - = 3 = / = %= & = ^ = | = < < = > > =	从右至左	二元运算符
	,	从左至右	二元运算符

从表 1 中可以看出,赋值运算符为右结合,二元运算符都满足左结合;条件运算符(即问号表达式) 为右结合;一元运算符和后缀运算符为右结合。这意味着 3 p + + 将被解释成 3 (p + +) 而非(3 p) + + 。

回到上述问题的讨论,既然二元加运算符和二元减运算符是左结合的,上述的计算结果应该没有什么问题。很遗憾,C 语言标准规定的只是运算符的结合顺序,而对于二元运算符两边操作数的求值顺序则未作定义。“对于二元操作符,要先对两个操作数求值(但没有特定顺序) 之后再进行运算”。因此,上述问题的答案已经揭晓:最终的结果取决于编译器特性。

如果使用的编译器总是从左向右解析表达式,则结果是:
问题一的答案是 result = ‘2’$ ‘4’= 0xFE;
问题二的答案是 result = 2 + 2 = 4 。

如果使用的编译器总是从右向左解析表达式,则结果是:
问题一的答案是 result = ‘4’$ ‘2’= 0x02 ;
问题二的答案是 result = 1 + 1 = 2 。

为了避免使用不同编译器而导致的程序结果差异,MISRA - C 提出了如下强制性规则。

规则 12. 2 :表达式的值必须在任何求值顺序下保持一致。

那么,什么时候会出现表达式的值不一致的情况呢?
MISRA - C 列出了几种可能,如自增运算符和自减运算符使用时、函数参数传递时、函数调用时等。
归纳起来,当表达式中的操作数(也可能是一个表达式) 能够影响某个共享的变量,而这个共享变量又可能导致其余操作数的值发生变化时,就需要对求值顺序保持警惕。典型的一个例子就是 i + (+ + i) ,此时+ + i 的操作会引起共享变量 i 的变化,而 i 的变化将直接影响到第一个操作数的值,于是不同的求值顺序导致了不同的求值结果。在问题 1 中,表达式的两边都使用了 uart_ GetChar ()操作(确切地说是通过该函数共享了同一个数据寄存器) ,所以求值顺序会影响结果。

解决此类问题的最好办法是把表达式重组,即将一个较为复杂的表达式分解成若干个简单的表达式,使运算符的多个操作数之间的耦合关系得以解除,由此保证求值的顺序。例如,可以将问题 1 和问题 2 改写成如下的形式。

问题一:
result = uart_GetChar () ;
result - = uart_GetChar () ;

问题二:
result = temp + + ;
result + = + + temp ;

为了防止表达式的求值结果和所期望的结果相悖,MISRA C 还提出了许多行之有效的表达式的书写规则。

规则 12. 1 (推荐) :应该减少表达式对 C 语言运算符优先级的依赖性。

这意味着在更多的情况下,应该用括号“() ”来保证运算顺序,而不依赖于 C 语言的运算符优先级,因为 C 语言某些运算符的优先级容易引起误解。

下面分析如下程序:
if (STATRegister & BUSYMask = = 0) {

```
    do_something ;
}
```

该程序的本意是读状态寄存器(STATRegister) 的值,如果其 BUSY 位是 0 (即被掩码 BUSYMask 选中) ,则做某些操作。但是,由于判等运算符“= = ”的优先级高于位与操作符“&”,实际的判断表达式变成了“(STATReg2ister & (BUSYMask = = 0)) ! = 0”。此时应该加入括号“() ”以保证判断表达式的正确性:

```
if ( (STA TRegister & BUSYMask ) = = 0)     {
    do_something ;
}
```

在程序中,容易出现混淆的地方,也应该通过括号“() ”来组织语句,如

```
if (a & & b | | c & & d)
```

应该改成:

```
if ( (a & & b) | | (c & & d) )
```

这将使得层次更加清晰,维护起来更加方便。

2 表达式的副作用

所谓的副作用,是指在表达式执行后对程序运行环境可能会造成影响。赋值语句、自增操作等都是典型的具有副作用的操作。此类操作关系到程序运行环境的改变,因此对有副作用的表达式需要格外小心。

规则 12. 3 :不允许将 sizeof 运算符作用于有副作用的表达式上。

试分析以下的代码:

```
int32_t     i ;
int16_t     j ;
j = sizeof (i = 1234) ;
```

本意是先将 1234 赋给 i ,再把 i 所占用的空间大小传给 j 。可是由于 sizeof 运算符只针对数据类型进行操作,所以“j = sizeof (i = 1234) ”实际上被替换成“j = sizeof(int32_t) ”。故表达式“i = 1234”的操作不会进行,这就带来了可能的隐患。正确的做法是将最后一句替换成:

```
i = 1234 ;
j = sizeof (i) ;
```

规则 12. 4 :逻辑运算(& & 和| |) 的右操作数不允许包含副作用。

在 C 语言的表达式中,部分代码可能不被求值。如果这些代码具有副作用,就会产生一些隐患。典型的例子出现在逻辑运算中:

```
if (ist rue | | do_something_with_side_effect s () )     {
    do_something ;
}
```

如果 ist rue 非 0 ,编译器认为表达式的值已经确定为真,从而不再进行后面的求值,于是有副作用的操作被忽略,影响了后继操作。为了避免出现这种问题,必须把较复杂的操作数放在逻辑运算符的左边,把简单表达式放在右边。如果两个表达式都比较复杂,应该先对某一个表达式求值,并将结果作为逻辑运算符的右操作数,如:

```
value = expression1 ;
```

```
if (expression2 | | value)     {
    do_something ;
}
```

MISRA C 中还有一些防止表达式运算结果出现歧义的规则。

规则 12. 5 :逻辑运算符的操作数必须是一个主表达式。(注:这里主表达式包括标识符、常量和括号括起来的表达式。)

规则 12. 6 (推荐) :逻辑运算符(& &、| | 和!) 的操作数必须为一个有效的布尔值,布尔值表达式不允许进行逻辑运算以外的操作。(注:这是为了防止误用。)

规则 12. 7 :不允许对有符号数进行位操作。(注: 这是为了防止结果的不确定性。)

规则 12. 8 :移位操作的右操作数只能在 0 和操作数的位数减 1 之间。(注:移位操作的右操作数就是移位的位数,比如一个 8 位的无符号整数,允许移位的位数范围是 0～7 ,这是为了防止未定义的操作。)

规则 12. 9 :不允许无符号性的表达式进行一元负运算符。(注:同规则 12. 8 。)

规则 12. 10 :不允许使用逗号表达式。(注:这是为了防止阅读混乱,另外,逗号表达式也可以用其他等价形式替代。)

规则 12. 12 :不允许对浮点型值进行位操作。(注:这是为了防止不同标准产生的差异性。)

规则 12. 13 (建议) :不允许在同一个表达式中混合使用+ + 和- - 。(注:这是为了防止阅读混乱,并防止出现歧义。)

3 小 结

在 C 语言编程中,大部分潜在错误来自于程序员对程序元素的误用或滥用,以及程序员对某些结构一厢情愿的理解。

编写代码时,没有必要太多地去追求所谓程序的“优雅”和“风度”,程序的价值不仅在于运行,还在于为后继的可能升级和维护提供一个参考。一个“优雅”的句式和写法也许能够让作者煞费苦心并乐此不疲,但同样也会让几个月或几年后的你或其他人伤透脑筋。在硬件资源日益丰富的今天,大部分程序员已经不需要像先辈那样去考虑如何缩减一个位的存储单元。随着项目规模的扩大,对程序可读性和可维护性的要求日益突出。程序的可读性和可维护性已经成为衡量程序价值的重要标准。

MISRA C 为嵌入式软件的可靠性提出了中肯的建议,一些嵌入式 C 编译器的开发商开始把 MISRA 出版物作为标准并予以支持。通过学习 MISRA C ,可以更多地了解 ANSI C 中的不确定因素,并在开发中尽可能避免误入“表达式失控”的“雷区”。

参考文献

1 MISRA C:2004 ,Guidelines for the use of the C language in critical systems. The Motor Indust ry Software Reliability As2 sociation ,2004

2 Harbison III. Samuel P , Steele J r. 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 林锐. 高质量 C + + / C 编程指南,2001

4 ISO/ IEC 9899 :1999. International Organization of Standardi2 zation , 1999

学习 MISRAC 之五:

准确的程序流控制

程序的执行流程是由条件判断、跳转和循环构成的,没有任何一个程序会缺少程序流的控制。那么像 if 、for 、while 、switch 等这些程序员无比熟悉的语句也存在隐患吗? 事实上,C 语言是很灵活的,这种灵活性给程序员编写代码带来了很多便利,但同时也带来了很多容易导致混淆的表达。这些表达完全符合 C 语言标准,但有时程序员也难以发现自己犯了错误,最终的结果是使程序进入错误的执行流程。即使程序员没有犯错误,但有些容易混淆的表达也会给其他人读懂程序带来困扰,使程序的维护变得困难。除此以外,有少量控制流程的方式还会产生不确定的运行结果,而这些结果也不容易被发觉。

如何使程序的流程控制清晰、准确,不产生混淆的表达呢? MISRA C 给出了很多的相关规定,使程序流的控制变得规范,避免产生各种混淆和不确定性,从而最大程度上减少程序流控制中的失误, 并使程序的维护更加容易。

下面从几个例子出发,讲述这些混淆是如何产生的,最后给出 MISRA C 关于程序流控制的相关规则,帮助读者规范编程的习惯。

1 容易混淆的表达方式

先来看这样两段代码:

```
uint8_t x ,y ;
…
if ( x = = y) {
    foo ( ) ;
}
uint8_t x ,y ;
…
if (x = y) {
    foo ( ) ;
}
```

在 C 标准中,条件语句需要的是布尔值,条件语句表达式的布尔值实际上是按照整型处理的,所以这两段代码在语法和逻辑上都没有任何问题。第一段代码判断 x 是否等于 y ,如果相等,调用 foo () 函数;第二段代码首先将 y 的值赋给 x ,然后判断 x 是否为 0 ,如果不为 0 ,调用 foo ()函数。这两段代码只相差一个等号,却使判断条件大不相同,程序的执行流程会出现很大差别。

相信读者在写程序的时候都碰到过将“= = ”这个判断语句误写成赋值语句“= ”的情况。那么面对这两个语句时,如何能快速准确地判断这是正确的还是程序员的失误呢? 当程序比较简单的时候,很容易判断,但当程序流程比较复杂的时候,可能花费大量时间还难以给出确定的答案,而这些地方极有可能是有错误的。

这样的混淆,事实上是可以轻松避免的,MISRA C 提出了如下强制性的规则。

规则 13. 1 :赋值表达式不能用在需要布尔值的地方。
按照 MISRA C 的标准,第二段代码应该写成:

```
uint8_t x ,y ;
…
x = y ;
if (x ! = 0) {
    foo ( ) ;
}
```

这样,当看到需要布尔值的地方出现了赋值表达式,
就可以立即判断这是一个错误。在这条规则下,如下的表达也是不允许的:

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

与这条规则类似,MISRA C 还提出了如下推荐的规则,来避免整型变量和布尔型的混淆。
规则 13. 2 (推荐) :判断一个值是否为 0 应该是显式的,除非该操作数是一个布尔值。
这条规则禁止了如下的表达:

```
uint8_t x ;
…
if ( x )
{…}
```

再来看一个例子:

```
uint8_t x ;
uint8_t a ,b ;
…
switch ( x ) {
    case 1 :
        a = b ;
    case 2 :
        a + = 2 ;
        break ;
    case 3 :
    …
}
```

同样,这段代码在语法和逻辑上也没有任何问题,编译器也不会给出任何错误或者警告。在程序执行中,当 x 等于 1 的时候,将 b 的值赋给 a ,然后将 a 加 2 ,退出;当 x 等于 2 的时候,直接将 a 的值加 2 ,接着退出。但这儿很可能是一段错误的代码,程序员的本意有可能是 x 等于 1 时,将 b 的值赋给 a ,当 x 等于 2 时,直接将 a 的值加 2 。
为了避免这样的混淆,MISRA C 提出了如下强制性的规则。

规则 15. 2 :所有非空的 switch 子句都应该以 break 语句结束。

按照这条规则,上面的程序应该写成:

```
switch ( x ) {
    case 1 :
        a = b ;
        break ;
    case 2 :
        a + = 2 ;
        break ;
    case 3 :
        …
}
```

或者:

```
switch ( x ) {
    case 1 :
        a = b ;
        a + = 2 ;
        break ;
    case 2 :
        a + = 2 ;
        break ;
    case 3 :
    …
}
```

MISRA C 中还有一些防止程序流控制中出现混淆的规则。

规则 13. 5 :for 语句中的 3 个表达式只能和循环控制相关。第一个表达式只能为循环变量赋初值,第二个表达式只能进行循环条件的判断,第三个表达式只能进行循环变量增(减) 值。

规则 13. 6 :for 循环中,循环变量只能在 for 语句的第三个表达式中修改,不允许在循环体中修改。

规则 13. 7 :布尔表达式的值必须是可以改变的。

例如,如下代码是不允许的:

```
uint8_t x ;
…
if ( x > = 0 )
…
```

错误在于该条件判断的结果始终为真。

规则 14. 1 :不能存在无法执行到的代码。

规则 14. 2 : 非空语句必须要么产生副作用(side-effect) (副作用是指表达式执行后对程序运行环境造成的影响。赋值语句、自
增操作等都是典型的具有副作用的操作。) ;**或者使程序流程改变。**
例如,下面的代码是不允许的:

```
…
x > = 3 ;
…
```

错误在于 x 和 3 比较的结果被丢弃了。

规则 14. 3 :一行中如果有空语句,那么该行只能有这条空语句,不能有别的语句,并且在这条空语句前不能有注释,注释必须在其后,用空格隔开。
例如,如下的代码都是不允许的:

```
while (port & 0x80 = = 0) {
; foo ( ) ; ①
/ * wait for pin * / ; ②
;/ * wait for pin * / ③
}
```

其中
的错误是除了空语句还有一条语句;
②的错误是在空语句前有注释;
③的错误是空语句与注释没用空格隔开。

规则 14. 8 : switch 、while 、do. . . while 和 for 语句的主体必须是复合语句(即用大括号包含) ,即使该主体只包含一条语句。
例如,如下代码是符合 MISRA C 标准的:

```
for ( i = 0 ;i < N_EL EMEN TS ; + + i )
{
    buffer [ i ] = 0 ;
}
```

规则 14. 9 :if 结构后面必须是一个复合语句(即用大括号包含) ,else 后面必须是一个复合语句(即用大括号包含) 或者另一个 if 语句。

规则 15. 1 : switch 语句的主体必须是复合语句(即用大括号包含) 。

规则 15. 2 :所有非空的 switch 子句都应该用 break 语句结束。

规则 15. 3 : switch 的最后一个子句必须是 default 子句,如果 default 中没有包含任何语句,那么应该有注释来说明为什么没有进行任何操作。

规则 15. 4 : switch 表达式不能是一个有效的布尔值。

例如,下面的代码是不允许的:

```
switch ( x = = 0 )
{…}
```

规则 15. 5 : switch 语句必须至少包含一个 case 子句。

2 导致混乱的表达方式

在 C 语言中,有一些表达方式可以使程序的代码量减少,但却会使程序的结构化程度降低,流程控制变得混乱,可读性大大降低。看下面一段代码:

```
if ( a > 0x02 )
{
    loop1 : b + = 1 ;
    if ( c > 0xA0 ) {
        goto loop3 ;
    }

    loop2 : a = 2 3 b ;
    c = a + b ;
    if ( c < 0x40) {
        goto loop1 ;
    }el se {
        goto loop2 ;
    }
}
…
loop3 : …
```

这段代码读起来很困难。实际编程时,程序员实现这段功能的代码自然不会这样写,但是当程序流程复杂的时候,各种看起来能使编程工作变得轻松的表达,例如 goto 、continue 等语句,却会使程序流程变得混乱,可读性降低,而隐藏其中的问题,很可能就无法发现了。

针对这种情况, MISRA C 给出了下面几条强制规则。

规则 14. 4 :不允许使用 goto 语句。

规则 14. 5 :不允许使用 continue 语句。

规则 14. 6 :循环体中最多只能出现一个 break 语句用于结束循环。

规则 14. 7 :函数只能有一个出口,这个出口必须在函数末尾。

规则 14. 10 : if . . . else if 结构必须由一个 else 子句结束。

当 if 语句后面有一个或者多个 else if 语句时,最后的一个 else if 必须有一个与之对应的 else 语句。如果只有一个 if 语句时,else 不是必须的。

3 不确定的执行结果

除了导致混淆和混乱的表达外,还有一些对浮点数的操作会导致不确定的结果。来看如下一段代码:

```
float32_t x ,y ;
… / 3 一些运算 3 /
if ( x = = y )
{ …}
```

if 的条件无法肯定什么情况为真。这是因为浮点数在计算机中无法用二进制精确表示,其运算总会存在舍入和切断误差,很多人看起来相等的结果,但计算机给出的两个浮点数并不相等,所以上面代码中 if 的主体语句什么情况执行是不确定的。MISRA C 给出了两条相关的规定来解决这一问题。

规则 13. 3 :不允许对浮点数进行相等或者不相等的比较,即使是非直接的比较也是不允许的。例如,如下非直接的比较也是不允许的:

```
float32_t x ,y ;
…
if ( x < = y )
{ …}
```

规则 13. 4 :for 循环的控制表达式不应包含浮点数类型。

4 小 结

好的代码,要安全可靠、有很好的可读性和可维护性。在 C 语言中,一些表达方式,可能会稍微减少程序员编程的工作量,但却会使程序的流程变得难以判断,其中的错误可能就无法发现。

按照 MISRA C 的标准来写代码,就可以避免程序流程产生混淆和混乱,排除其中的不确定因素,使程序真正按照程序员设想的工作,并使代码更清晰易懂,真正实现安全可靠,并具有良好的可读性和可维护性。

参考文献

1 MISRA C:2004 ,Guidelines for the use of the C language in critical systems. The Motor Indust ry Software Reliability As2sociation ,2004

2 Harbison III. Samuel P , Steele J r. 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 Les Hatton. The MISRA C Compliance Suite The next step , Oakwood Computing. http :/ / www. misra c2. com/

4 ISO/ IEC 9899 :1999. International Organization of Standardi2zation , 1999

学习 MISRAC 之六:

构建安全的编译环境

预处理是编译环境处理 C 程序的第一个环节,但往往最先被程序员忽略。这份看似只是由编译环境做的简单工作,其实也是机关重重。通过介绍 MISRA C 与预处理相关的规则,希望读者能够更准确地认识编译器的预处理过程,避免出错。无论是自定义函数还是由编译环境提供的标准库函数,如果使用不当,都会存在安全隐患。

能不能保证函数被正确的定义、声明和调用,关乎到整个程序的成败。这里介绍 MISAR C 中涉及函数部分有代表性的规则,并试图分析制定这些规则的出发点,以帮助读者构建更为安全的编译环境。

1 函数的定义和声明

1. 1 在哪里定义

读者也许会有这样的经历:在一个头文件中定义了一个变量,又让这个头文件被多个源文件引用。这时编译器会“报怨”说重复定义了同一个函数。出错的原因与其说是粗心,不如说是一个习惯的问题。

规则 8. 5 :头文件中不允许包含对象或函数的定义。

当源文件包含某一头文件时,预处理器会将头文件的内容在包含指令处展开。显然,在头文件中函数的定义会在其他源文件中一模一样的出现,导致函数被重复定义。解决这一问题的关键是明确一个概念:所有可执行的代码或者对象和函数的定义都应在. C 的源文件中,头文件中只能存在其声明。具体的做法是:为全局变量的声明增加 extern 修饰符,并在相应的. C 源文件中定义对象或函数。

例如,在 Globle. h 文件中仅声明变量 GCounter 。

```
/ * 在 Globle. h 中 * /
extern uint32_t GCounter ;
.....
```

而在. C 文件中定义变量 GCounter :

```
/ * 在 GlobleVariables. C 中* /
uint23_t GCounter ;
.....
```

这样,就可以在所有需要用到全局变量的地方直接引用”Globle. h”头文件了。不过也有一些程序员喜欢采取以下的做法:

```
/ * 在 Globle. h 中* /
# ifdef GLOBL ES
# define EXT
# else
# define EXT extern
# endif
```

```
EXT uint32_t GCounter ;
……
/ *  在 GlobleVariables. C  中* /
# define GLOBL E
# include“Globle. h”
……
```

```
/ *  在其他 C  文件中* /
# include“Globle. h”
# include“MyLib. h”
        ……
```

这样做的好处是只需要维护“Globle. h”就可以维护所有全局变量。其他的文件中,直接包含“Globle. h”就可以使用这些全局变量了。上述的两种做法是等价的,读者
可选择任意一种方式。

1. 2　用好编译器的检查功能

在 MISRA C 制定关于函数编程规则的背后,有一条很重要的思想:要充分利用编译器的类型检查功能来提高函数的可靠性。这里的类型检查包括在函数定义和调用时对函数参数和返回值的类型检查。

规则 8. 1 :函数必须声明原型,在函数定义和调用时原型必须可见。

首先明确一下什么是原型声明。在科尼汉和里查(K& R) 的著名著作《C 程序设计语言(第二版) 》的前言中提到:“C 不是一种强类型语言,但随着它的发展,其类型检
查机制已得到了加强……在这个方向上,新的函数声明方式是迈出的另外一步。”

这里“新的函数声明方式”指的就是原型声明。原型声明是标准 C 语言中出现的概念,它可以提供更多关于函数参数的信息。在原型声明中,函数的参数要在声明时
指定参数名和类型;而非原型声明,参数的类型可以缺省,被忽略的参数声明默认为 int 型。

请看下面的声明:

```
int f (int i , long j) { ……} (原型声明)
int f (i ,j) int i ; { ……} (非原型声明)
```

要求程序使用原型声明函数,主要是希望可以利用编译器检查函数调用时数据类型的一致性。如果调用函数时,没有进行原型声明,则编译器不会检查出函数形式参数与调用参数的不一致。请看下面这段程序:

```
double square (x)
double x ;
{
……
}
```

调用时:

```
long func (i)
long i ;
{     return square (i) ;
}
```

函数 square () 的形式参数类型是 double 型,但实际调用时,调用参数是 long 类型。因为没有进行原型声明,编译器不需要对此给出警告,结果在没有出错信息的情况下,函数返回了一个不正确的值。

现在将这段程序改写为原型声明:

```
double square (double x)
{     …
}
```

调用时:

```
long func (i)
long i ;
{     return square (i) ;
}
```

这里,编译器会检查出函数 square 的实际调用参数和形式参数类型不符,并且会将实际参数转换成相应的形式参数的数据类型。这样,参数 i 就在程序员不知情的情
况下被编译器自动转换为 double 类型,函数返回正确值。

了解了原型调用的一些机制,下面的问题就是如何操作才能保证每个函数调用都使用原型调用,也就是说,要使原型声明对于函数定义和调用都“可见”。简单的方法是:每一个外部函数都在头文件中有一个唯一的原型声明;需要调用此外部函数的源文件要包含这一头文件,保证调用时由原型控制(原型对于“调用”可见) ;同时,函数定义所在的源文件也包含这一头文件,以便编译器可以检查原型声明和其定义相匹配(原型对于“定义”可见) 。

使用原型调用除了可以帮助检查参数的一致性,还可以使“编译器产生更为有效的函数调用序列”。
为了配合编译器对函数参数的检查,程序员应牢记规则 16. 1 。

规则 16. 1 :不允许定义参数数量不确定的函数。

标准库函数 printf ( ) 深受许多程序员的喜爱,因为 printf () 允许不确定的参数数量,用起来很方便。但是,参数数量的不确定很可能会造成编译器无法检查函数调用时的参数一致性。对于像 printf () 这种使用广泛的标准库函数,编译器提供了一些合适的调用机制。但程序员必须明确,编译器无法保证对用户自行定义的参数数量不确定的函数进行数据类型检查。因此,MISRA2C 不允许用户冒险去定义新的参数数量不确定的函数。

2 函数的调用和标准库函数

程序员应该清楚,嵌入式应用开发中系统的资源往往十分有限,在程序开发上会有特殊的限

制。比如,在 RAM 空间的使用上往往会捉襟见肘。像早期的 PC 机程序员一样,对 RAM 空间的使用可以用“吝惜”来形容,往往因可以使程序少占用几个字节的 RAM 而大做文章。

一个典型的例子就是递归函数的调用。递归函数的代码紧凑,且容易理解,很受 C 程序员的推崇。但递归函数的一个缺点就是:占用 RAM(这里主要是指的栈空间)

资源太多。对于嵌入式系统来说这是尤为严重的问题。一旦递归调用的层数过多,就会出现栈空间不足的情况。唯一可以避免该情况发生的方法就是能够预先估计出最大的递归调用层数,从而算出最大栈空间。遗憾的是,很多情况下程序员根本没法做出估计,这时系统中的递归函数成为一个巨大的隐患。MISRA C 从系统安全角度考虑,选择了最为安全的做法,不准使用递归调用。

规则 16. 2 :函数不得调用本身,无论是直接的调用,还是间接的调用。

一般来说,标准库函数是很好用的。它的定义和使用都很清晰,尤其是像 printf ( ) 这样的函数,对于程序员的调试工作帮助很大。但某些库函数的使用也可能会造成问题。要尽量安全地使用库函数,需要注意三个方面的问题。

①要保证库函数头文件中的宏、标识符和函数的定义不受干扰。

规则 20. 1 :不得定义、重新定义或是取消定义标准函数库中的标识符、宏和函数。

②要按照正确的方法使用库函数。库函数对参数的类型、数值都有很明确的要求,只有传递给库函数正确的参数,才能保证结果的正确性。

规则 20. 3 :必须检查传递给库函数的数值的有效性。

③避免使用可能有问题的库函数或者其结果。比如很多库函数都会通过一个叫做 errno 的变量为非零值来表示执行失败。但是,由于没有强制库函数在执行成功后将 errno 清零,一个非零的 errno 有可能是因为当前库函数执行失败了,也有可能是因为之前某个库函数没有正确执行。因此,完全依赖 errno 来判断库函数的执行成功与否是不可靠的。

规则 20. 5 :不得使用错误指示符 errno 。

可能带来问题的库函数还有很多,MISRA C 为此做了一份总结。

规则 20. 4 :不得使用动态堆空间分配。

规则 20. 6 :不得使用库函数< stddef . h > 中的宏 offsetof

规则 20. 7 :不得使用 longjmp 函数中的宏 setjmp

规则 20. 8 :不得使用信号处理函数< signal . h >

规则 20. 9 :不得用输入/ 输出库函数< stdio. h > 来产生代码

规则 20. 10 :不得使用标准库< stdlib. h > 中的库函数 atof 、atoi 和 atol

规则 20. 11 :不得使用标准库< stdlib. h > 中的库函数 abort 、exit 和 system

规则 20. 12 :不得使用标准库< time. h > 中的时间处理函数

3 预处理———看似简单的第一步

预处理是编译器处理程序的第一步。预处理会在编译器编译程序代码前做一些准备工作,最为常见的工作是处理文件包和宏定义(分别对应# include 和# define 两个
预处理指令) 。
预处理器并不对源代码做编译,只是进行一些转换工作,例如将文件或宏展开等。很多程序员认为这种类似复制、粘贴的活没什么了不起,也就放松了对预处理工作的检
查。其实,很多程序的失败就从这看似简单的第一步开始。
宏定义是最常见的预处理指令之一。MISRA C 关于宏定义有一些很有代表性的规则。

3. 1 小括号———一个也不能少

当程序中多处出现同一个数值的时候,程序员就会想起使用宏。宏定义最大的好处就是使一些常量集中起来,修改其值只需要修改一次,大大提高了程序的可维护性。
编译器对宏的处理原则比较简单,宏定义只对程序文本起作用。一个经典的例子是:
define abs (x) (x > = 0) x : - x

显然,程序员希望求变量 x 的绝对值。但是,下面的调用会是什么结果呢?
abs (a - b) ;
展开后为(a - b > = 0) a - b : - a - b 。显然这里的- a - b
并不是程序员想要的结果,原因是变元 x 两边没有加小括号。那么在 x 加上括号以后呢?
define abs (x) ((x) > = 0) (x) : - (x)
如果此时是
abs (a) + 1 ;
展开后是(a > = 0) a : - a + 1 ,也不是我们想要的结果。
看来还要把整个宏定义加上括号,这样才能得到安全可靠的宏定义:
define abs (x) (((x) > = 0) (x) : - (x))
为了防止宏展开后因缺少括号而存在的优先级错误问题,MISRA C 有如下规定。

规则 19. 10 :在函数式宏定义中,任何一个参数都应加上小括号,除非是在# 或# # 运算符中。

3. 2 宏定义不是函数

规则 19. 7 (推荐) :应优先考虑使用函数而非函数式宏定义。

利用类似函数式的宏定义来取代函数调用,是一个常用的技巧。这样做有很多好处,主要是能够提高程序的运行速度。MISRA C 从代码安全的角度制定这一规则,主要有两点考虑。一是宏定义不能像函数调用那样提供参数类型检查,错误的变元类型无法得到纠正,运行的结果就可能不正确。二是宏定义中的变元可能会多次求值,当变元表达式带有副作用时,就会出现问题。例如:

define SQUARE(x) ((x) 3 (x))
当有如下语句时:
a = 3 ;
b = SQUARE(a + +) ;

程序员肯定希望得到 b = 9 和 a = 4 的结果,可实际上的结果却是 b = 12 和 a = 5 ,这是为什么呢?
如果考虑到宏展开只是做文本的展开,那么上面的预处理结果应该是:
a = 3 ;
b = (a + +) 3 (a + +) ;
很明显,这里 a + + 运行了两次,运行后 a = 5 。至于 b ,其结果应该是 b = 3 ×4 = 12 。
现在,读者应该可以看出来类似函数的宏展开并不完全和函数一样。考虑到系统可靠性是我们所关注的,上面的工作还不如直接用函数来完成。多数情况下,函数的运
行速度应该让位于其结果的正确性。
关于宏定义还有很多有趣的问题可以讨论,这里就不一一赘述了。

结 语

至此, 关于 MISRA C: 2004 的学习暂告一段落。

MISRA C:2004 有 141 条规则,在 6 期的《学习园地》栏目中,列举和解释了其中有代表性的规则,大约二分之一,且尽量使每篇文章都能涵盖 MISRA C 规范的一个重要方向,以便使读者了解到 MISRA C 的概貌和主导思想。

由于 MISRA C:2004 一书中关于每条规则的解释很少,很多例子是在我们理解的基础上加的,可能存在着错误或偏差,欢迎大家和我们共同讨论。

通过这 6 期介绍,希望大家能够意识到:C 是一门并不容易掌握的语言。作为嵌入式工程师,多数人只是某应用领域的专家,对于 C 语言编程,有个从“业余”到“专业”
的过程,学习和参考 MISRA C 可以帮助他们编写出更安全、更“专业”的代码。尤其对于一些安全性要求很高的小系统,MISRA C 是非常合适的安全编程规范。这里小系统指开发团队的规模小,甚至全部工作都是由一个人完成的系统。

尽管手册式的编程规范很难引起读者的兴趣,MIS2RA C:2004 还是值得仔细品味的。经验对于程序员来说是一笔财富。MISRA C 的编写专家们大都来自于汽车工业及相关软件公司,他们有着丰富的汽车安全方面的知识和软件开发经验,MISRA C:2004 很大程度上是他们对如何提高 C 软件可靠性的经验总结。对于大多数程序员来说,仔细研读这份经验总结可以少

走很多弯路,同时提高自身的编程素养。

目前,“嵌入式”还不是一个学科或专业,也许永远也不会是一个独立的学科。然而,各行各业都需要嵌入式系统,都有义务推动“嵌入式学科”(如果能这样表述的话) 的发展。例如,便携类应用就推动了嵌入式系统低功耗技术的发展。MISRA C 从汽车工业软件可靠性角度,对 C 语言的使用做出种种限制,使之成为汽车工业的行业标准,并被其他对可靠性要求高的行业采纳,这是汽车行业对嵌入式领域的贡献。其他行业的嵌入式工程师也应该有责任、有能力在借鉴其行业相关技术、规约的基础上,从不同角度推动嵌入式技术的全面发展。

参考文献

[1] MISRA C:2004 , Guidelines for the use of the C language in critical systems. The Motor Indust ry Software Reliabili2 ty Association ,2004.

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

[3] Harbison III. Samuel P ,Steele J r. Guy L. C 语言参考手册. 邱仲潘,等译. 第 5 版. 北京:北京机械工业出版社,2003.

[4] Les Hatton. The MISRA C Compliance Suite The next step , Oakwood Computing. http :/ / www. misra c2. com.

[5] ISO/ IEC 9899 :1999. International Organization of Standardization , 1999.