第四章　单片机 C 语言

终于结束了晦涩枯燥的第三章，我自己也长吁了一口气，现在我真的是非常同情那些教专业基础课或者工程数学的老师，这里真诚的说一声："辛苦你们了"。

这一章要轻松不少，相信就算是刚走出校门的雏鸟，多少都有一定的 C 语言基础，大学好象都要过一个计算机等级考试，所以 C 语言自然是逃不了。你不要指望我给你讲述 C 语言原理和指令这类基础知识，我更不会给你讲述 C++那些面对对象编程的高级编程方法，这一章讲述的内容都是 C 语言在单片机上应用会遇到的一些有意思的现象，让你知道 C 在单片机上是怎么工作的。

当然也会告诉你一些 C 的经验技巧，这些对提升你的单片机程序能力还是有一定作用的。

4.1. 单片机 C 语言

早期单片机编程是没有 C 语言支持的，都是汇编甚至是二进制的机器码，随着电脑技术的突飞猛进，单片机编程不再安于汇编的一亩三分地，也向着 C 语言的方向进发。理论上讲单片机实现 C 语言编程不存在丝毫问题，毕竟和电脑是同根生，于是一批专业或非专业、有着利益目的或无利益目的的工程师开始了这方面的努力。

和电脑最大的不同是单片机种类繁多，不象电脑只有那么几种芯片，而且电脑 CPU 的发展遵循着一定的规则，不同 CPU 要求做到指令兼容，单片机做这样的要求显然不现实，厂商不可能接受都遵循制定标准设定 MCU 的要求。虽然单片机种类繁多，但大部分单片机还是会采用通用构架进行设计，毕竟遵循一定标准可以不用厂商自己去完成指令系统、编译工具等繁琐工作，所以市面上流行的单片机内核其实并不多，不少八位的单片机都采用 51 内核，高端的 MCU 内核更是集中在 ARM/MIPS…这几种当中。

厂商设计的 MCU 通常都会沿用某一种构架，也就是厂商产品目录中的 xx 系列，这样做厂商可以节省开发成本，一套编译器可以为一个甚至多个系列的 MCU 所用，这样新设计 MCU 或编译器有问题也可以在日后进行改进，如果弄成一种 MCU 就对应一套编译器的方式，神仙也会疯掉。厂商为了占领更多的市场，自然就会依据市场需求针对 MCU 推出 C 的编译器，不过这种做法所退出的 C 编译器质量局限于厂商自己技术能力，通常说这类编译器可以用，但不要期望有着很高的效率。如果是流行面广的内核，会有另外一种方式，就是专业的软件公司针对这种内核的指令系统开发 C 编译器，象 KEIL C 就是一例，这种软件公司在编译方面经验丰富，所以他们做出来的编译器效率方面相当不错，只要是他们编译器支持的内核，就很容易让编译器支持。软件公司推出的 C 编译器虽然好，但要钱，有免费的版本可限制多多，技术世界从来不缺少活雷锋，GCC 这样的组织让免费获取 C 编

译器成为了现实，不过这类组织所支持的对象只能是内核为ARM/MIPS…的高端通用MCU。

想要做好单片机的C编译器则必须具备这两个条件：一是熟悉MCU的硬件资源和指令系统，二是熟悉C语言，两者缺一不可，否则是做不出一个优质高效的单片机C编译器的。编译器的工作就是将用C写的代码按一定规则转换成汇编指令，这样程序员面对的是接近自然语言的C代码，对程序的结构控制、含义理解等会容易不少。由于转换操作依赖编译器，虽然一个编译器需要经过大量测试才会推出，但测试无法涵盖所有的编程可能，这样编译器并不能保证可靠性为百分百，一旦有错误产生，调试会麻烦许多，毕竟错误不是程序员而是编译器产生的，在C语言层面会让错误弄得一头雾水，当然程序员对C和汇编都很熟悉的话还是可以通过查看汇编代码的方式来查找编译器错误。

同电脑的C相比单片机的C编程存在自己的特点：电脑用C写程序奉行的是硬件无关的原则，程序员只要了解C的语法就可以，就是深入到驱动程序层面也只需要了解驱动程序的接口就够，单片机则不然，C只是让程序员面对的代码不再是汇编格式，程序编写依然还是要了解硬件特性，只是将原本由汇编写的硬件控制代码改成了C的语法格式；为了最大程度的利用单片机的各种指令，单片机的C编译器同电脑的C编译器相比可能会有多不同，比如对某些C语法做出修改，象KEIL C对51系列的单片机就多出了位变量的定义和操作的语法；单片机结构要比电脑简单，所提供的资源也要少许多，希望能支持C编程也主要是为了让程序结构简单明晰，所以C的控制流程语法就已经够用，并不需要象电脑一样在标准C的基础上继续类似C++各种改进。但也不是绝对，现在各种嵌入式平台也还是尽力向电脑方向靠拢，这样做我的理解是众人拾柴火焰高，嵌入式已经发展到了需要许多人合作才能实现的阶段。

单片机用C编程便捷性无疑是大为提高，可当用C实现对单片机支持后新问题出现，这就是目前我们国家的现状是做单片机的大多是电子信息类的专业出生，在学习阶段以控制方面的知识为主，C只是做为辅助课程出现，更不用说软件工程之类的课程，这样当这部分人由学生转入工作时容易写出汇编C代码，就是语法是C而程序风格和思路是汇编，在计算机专业出身的人眼里看为垃圾代码，如果本身是计算机专业出身去写单片机C程序又会面临电子专业基础不足的问题。这一章我会告诉习惯汇编的单片机程序员一些C的经验和技巧，相信通过这章的学习会让你对C的单片机编程有更深的认识。

4.2. for()和while()循环

不同程序员都有自己的编程风格，让一个程序员用代码实现死循环，C出身的程序员习惯会用for(;;)和while(1)，而汇编出身的程序员则习惯用goto loop_label，风格上的差异都可以接受，但如果深入到代码效率层面就会发现不同的循环方式效率也会存在差别。

用C语言写出循环100次的九种不同实现方式：

```
void MCU_CTest(void)
```

```
{
  unsigned long i;
  unsigned long vTemp;
  //------loop mode 1-----
  //------loop mode 2-----
  //------loop mode 3-----
  //------loop mode 4-----
  //------loop mode 5-----
  //------loop mode 6-----
  //------loop mode 7-----
  //------loop mode 8-----
  //------loop mode 9-----
  //------end flag------
  vTemp=0;
}
```

方式一C 代码：

```
  vTemp=0;
  for(i=0;i<100;i++)
  {
    vTemp=vTemp+i;
  }
```

方式一汇编代码说明：

```
  0x1dc0:  MOV    R1,#0        ;vTemp=0，第一种循环
  0x1dc4:  MOV    R0,#0        ;i=0，循环次数清零
  0x1dc8:  CMP    R0,#0x64     ;将 i 和 100 比较
  0x1dcc:  BCS    0x1de4       ;i 大于等于 100 则跳到地址 0x1de4，结束循环
  0x1dd0:  B      0x1ddc       ;跳转到地址 0x1ddc 好进行 vTemp=vTemp+i
  0x1dd4:  ADD    R0,R0,#0x1   ;i++
  0x1dd8:  B      0x1dc8       ;跳转到地址 0x1dc8 好判断 i 是否小于 100
  0x1ddc:  ADD    R1,R1,R0     ;vTemp=vTemp+i
  0x1de0:  B      0x1dd4       ;跳转到地址 0x1dd4
  0x1de4:  MOV    R1,#0        ;vTemp=0，第二种循环开始
```

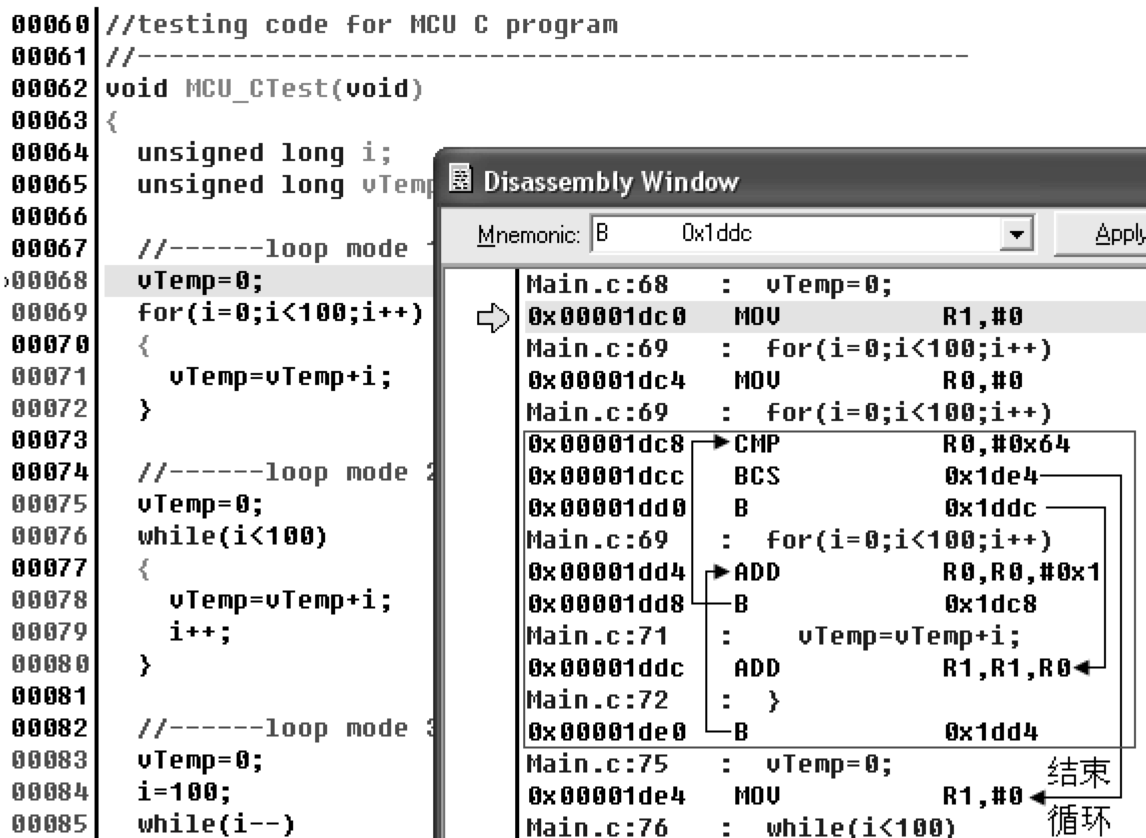

图 4.21.-1 ARM 汇编结果示意图一

方式一汇编代码执行流程：

```
0x1dc0:  MOV   R1,#0        ;vTemp=0，第一种循环
0x1dc4:  MOV   R0,#0        ;i=0，循环次数清零
0x1dc8:  CMP   R0,#0x64     ;将 i 和 100 比较，此时 i 为 0，第一次循环
0x1dcc:  BCS   0x1de4       ;比较结果不用跳转
0x1dd0:  B     0x1ddc       ;跳转到地址 0x1ddc
0x1ddc:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1de0:  B     0x1dd4       ;跳转到地址 0x1dd4
0x1dd4:  ADD   R0,R0,#0x1   ;i++
0x1dd8:  B     0x1dc8       ;跳转到地址 0x1dc8
0x1dc8:  CMP   R0,#0x64     ;将 i 和 100 比较，此时 i 为 1，第二次循环
0x1dcc:  BCS   0x1de4       ;比较结果不用跳转
0x1dd0:  B     0x1ddc       ;跳转到地址 0x1ddc
0x1ddc:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1de0:  B     0x1dd4       ;跳转到地址 0x1dd4
0x1dd4:  ADD   R0,R0,#0x1   ;i++
```

```
0x1dd8:  B      0x1dc8       ;跳转到地址 0x1dc8
0x1dc8:  CMP    R0,#0x64     ;将 i 和 100 比较，此时 i 为 2，第三次循环
……
```

方式一执行一次循环需要 7 条指令，其中真正跳转 3 次。

方式二 C 代码：

```
vTemp=0;
i=0;
while(i<100)
{
  vTemp=vTemp+i;
  i++;
}
```

方式二汇编代码说明：

```
0x1de4:  MOV    R1,#0        ;vTemp=0，第二种循环
0x1de8:  MOV    R0,#0        ;i=0，循环次数清零
0x1dec:  NOP                 ;空操作
0x1df0:  CMP    R0,#0x64     ;将 i 和 100 比较
0x1df4:  BCS    0x1e04       ;i 大于等于 100 则跳到地址 0x1de4，结束循环
0x1df8:  ADD    R1,R1,R0     ;vTemp=vTemp+i
0x1dfc:  ADD    R0,R0,#0x1   ;i++
0x1e00:  B      0x1df0       ;跳转到地址 0x1df0
0x1e04:  MOV    R1,#0        ;vTemp=0，第三种循环开始
```

```
1074  //------loop mode 2
1075  vTemp=0;
1076  i=0;
1077  while(i<100)
1078  {
1079    vTemp=vTemp+i;
1080    i++;
1081  }
1082
1083  //------loop mode 3
1084  vTemp=0;
1085  i=100;
1086  while(i)
1087  {
1088    vTemp=vTemp+i;
1089    i--;
1090  }
1091
```

```
0x00001de0   B            0x1dd4
Main.c:75    :  vTemp=0;
0x00001de4   MOV          R1,#0
Main.c:76    :  i=0;
0x00001de8   MOV          R0,#0
Main.c:77    :  while(i<100)
0x00001dec   NOP
Main.c:77    :  while(i<100)
0x00001df0   CMP          R0,#0x64
0x00001df4   BCS          0x1e04
Main.c:79    :    vTemp=vTemp+i;
0x00001df8   ADD          R1,R1,R0
Main.c:80    :    i++;
0x00001dfc   ADD          R0,R0,#0x1
Main.c:81    :  }
0x00001e00   B            0x1df0
Main.c:84    :  vTemp=0;        退出循环
0x00001e04   MOV          R1,#0
```

图 4.2.-2 ARM 汇编结果示意图二

方式二汇编代码执行流程比方式一要简单，执行阴影部分循环只需要 5 条指令，其中真正跳转 1 次。

为节约篇幅其它方式只将汇编代码整理出来，不贴图加以说明。

方式三 C 代码：

```
vTemp=0;
i=0;
while((i++)<100)
{
  vTemp=vTemp+i;
}
```

方式三汇编代码说明：

```
0x1e04:  MOV   R1,#0        ;vTemp=0，第三种循环
0x1e08:  MOV   R0,#0        ;i=0，循环次数清零
0x1e0c:  NOP                ;空操作
0x1e10:  MOV   R2,R0        ;将 i 的值用中间变量保存起来
0x1e14:  ADD   R0,R0,0x1    ;i++
0x1e18:  CMP   R2,#0x64     ;未加之前的 i 和 100 进行比较
0x1e1c:  BCS   0x1e28       ;大于等于 100 跳转到地址 0x1e28，结束循环
0x1e20:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1e24:  B     0x1e10       ;跳转到地址 0x1e10
0x1e28:  MOV   R1,#0        ;vTemp=0，第四种循环开始
```

方式三汇编代码执行阴影部分循环需要 6 条指令，其中真正跳转 1 次。

方式四 C 代码：

```
vTemp=0;
i=0;
do
{
  vTemp=vTemp+i;
}while((i++)<100);
```

方式四汇编代码说明：

```
0x1e28:  MOV   R1,#0        ;vTemp=0，第四种循环
0x1e2c:  MOV   R0,#0        ;i=0，循环次数清零
0x1e30:  NOP                ;空操作
0x1e34:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1e38:  MOV   R2,R0        ;将 i 未加之前的值保存起来
0x1e3c:  ADD   R0,R0,#0x1   ;i++
0x1e40:  CMP   R2,#0x64     ;将未加之前的 i 和 100 进行比较
0x1e44:  BCC   0x1e34       ;小于 100 跳转到地址 0x1e34
0x1e48:  MOV   R1,#0        ;vTemp=0，第五种循环开始
```

方式四汇编代码执行阴影部分循环需要 5 条指令，其中真正跳转 1 次。

方式五 C 代码：

```
vTemp=0;
i=0;
do
{
  vTemp=vTemp+i;
  i++;
}while(i<100);
```

方式五汇编代码说明：

```
0x1e48:  MOV   R1,#0        ;vTemp=0，第五种循环
0x1e4c:  MOV   R0,#0        ;i=0，循环次数清零
0x1e50:  NOP                ;空操作
0x1e54:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1e58:  ADD   R0,R0,#0x1   ;i++
0x1e5c:  CMP   R0,#0x64     ;将未加之前的 i 和 100 进行比较
0x1e60:  BCC   0x1e54       ;小于 100 跳转到地址 0x1e54
0x1e64:  MOV   R1,#0        ;vTemp=0，第五种循环开始
```

方式五汇编代码执行阴影部分循环需要 4 条指令，其中真正跳转 1 次。

方式六 C 代码：

```
vTemp=0;
i=100;
while(i—)
{
```

```
  vTemp=vTemp+i;
}
```

方式六汇编代码说明：

```
0x1e64:  MOV   R1,#0        ;vTemp=0，第六种循环
0x1e68:  MOV   R0,#0x64     ;i=100，循环次数设置为 100
0x1e6c:  NOP                ;空操作
0x1e70:  SUB   R2,R0,#0x1   ;i--，减的结果放到到中间变量中
0x1e74:  MOV   R0,R2        ;i--，将减得的结果取回
0x1e78:  CMN   R2,#0x1      ;将 i 和 1 进行比较
0x1e7c:  BEQ   0x1e88       ;等于跳转到地址 0x1e88，结束循环
0x1e80:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1e84:  B     0x1e70       ;跳转到地址 0x1e70
0x1e88:  MOV   R1,#0        ;vTemp=0，第七种循环开始
```

方式六汇编代码执行阴影部分循环需要 6 条指令，其中真正跳转 1 次。

方式七 C 代码：

```
vTemp=0;
i=100;
while(i)
{
  vTemp=vTemp+i;
  i—;
}
```

方式七汇编代码说明：

```
0x1e88:  MOV   R1,#0        ;vTemp=0，第七种循环
0x1e8c:  MOV   R0,#0x64     ;i=100，循环次数设置为 100
0x1e90:  NOP                ;空操作
0x1e94:  CMP   R0,#0x0      ;将 i 和比较
0x1e98:  BEQ   0x1ea8       ;比较结果相等跳转到地址 0x1ea8，结束循环
0x1e9c:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1ea0:  SUB   R0,R0,#0x1   ;i--
0x1ea4:  B     0x1e94       ;跳转到地址 0x1e94
0x1ea8:  MOV   R1,#0        ;vTemp=0，第八种循环开始
```

方式七汇编代码执行阴影部分循环需要 5 条指令，其中真正跳转 1 次。

方式八 C 代码：

```
vTemp=0;
i=100;
do
{
  vTemp=vTemp+i;
}while(i--);
```

方式八汇编代码说明：

```
0x1ea8:  MOV   R1,#0         ;vTemp=0，第八种循环
0x1eac:  MOV   R0,#0x64      ;i=100，循环次数设置为 100
0x1eb0:  NOP                 ;空操作
0x1eb4:  ADD   R1,R1,R0      ;vTemp=vTemp+i
0x1eb8:  SUB   R2,R0,#0x1    ;i--，减的结果放到到中间变量中
0x1ebc:  MOV   R0,R2         ;i--，将减得的结果取回
0x1ec0:  CMN   R2,#0x1       ;将 i 和 1 比较
0x1ec4:  BNE   0x1eb4        ;不相等跳转到地址 0x1eb4
0x1ec8:  MOV   R1,#0         ;vTemp=0，第九种循环开始
```

方式八汇编代码执行阴影部分循环需要 5 条指令，其中真正跳转 1 次。

方式九 C 代码：

```
vTemp=0;
i=100;
do
{
  vTemp=vTemp+i;
  i—;
}while(i);
```

方式九汇编代码说明：

```
0x1ec8:  MOV   R1,#0         ;vTemp=0，第九种循环
0x1ecc:  MOV   R0,#0x64      ;i=100，循环次数设置为 100
0x1ed0:  NOP                 ;空操作
0x1ed4:  ADD   R1,R1,R0      ;vTemp=vTemp+i
0x1ed8:  SUB   R0,R0,#0x1    ;i--
0x1edc:  CMP   R0,#0         ;将 i 和 0 进行比较
0x1ee0:  BNE   0x1ed4        ;比较结果不相等跳转到地址 0x1ed4
```

```
0x1ee4:  MOV   R1,#0        ;vTemp=0，所有循环结束
```

方式九汇编代码执行阴影部分循环需要 4 条指令，其中真正跳转 1 次。

对比九种不同循环方式的汇编代码，可以看出方式一的汇编指令最多，执行一次循环所耗费的时间最长，方式五和方式九的汇编指令最少，执行一次循环所耗费的时间也最短。这些循环方式效率满足 do while() > while() > for()的规律，而在实际中这三种 C 语言循环控制方式使用频率刚好相反，do while() < while() < for()。刚接触 C 语言编程的新人，最喜欢用的就是 for()方式，殊不知这种方式效率最低，所以如果是用 C 语言进行单片机编程，进入到熟练阶段后一定注意循环方式对代码效率的影响。

有兴趣的朋友可以仔细对比这几种方式的汇编代码，从中你可以找出编译器对 C 代码进行编译的规律：基本上按照 C 语言的顺序对应编译，对 i++/ i—这类先用再加的特殊语句是先将 i 的值取出来放到中间变量里面，然后将 i 进行加减处理，循环体中的 i 用中间变量进行替代。

4.3. 循环里的 i++与 i—

循环里面 i++和 i—的使用效果上也会有一些区别，上一节中我们列举了九种 C 语言的循环实现方式，其中方式五和方式九的效率最高，这两种方式一个采用的是 i++，另一个采用 i—，从产生的汇编指令看两者有着相同的效率，好象没有什么区别？别着急，接下来我让你相信确实有区别，而且明白区别是什么原因造成的。

方式五：

```
vTemp=0;
i=0;
do
{
  vTemp=vTemp+i;
  i++;
}while(i<100);
```

循环部分汇编代码

```
0x1e54:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1e58:  ADD   R0,R0,#0x1   ;i++
0x1e5c:  CMP   R0,#0x64     ;将未加之前的 i 和 100 进行比较
0x1e60:  BCC   0x1e54       ;小于 100 跳转到地址 0x1e54
```

方式九：

```
vTemp=0;
```

```
i=100;
do
{
  vTemp=vTemp+i;
  i—;
}while(i);
```

循环部分汇编代码

```
0x1ed4:  ADD   R1,R1,R0      ;vTemp=vTemp+i
0x1ed8:  SUB   R0,R0,#0x1    ;i--
0x1edc:  CMP   R0,#0         ;将 i 和 0 进行比较 *这条指令可以不要
0x1ee0:  BNE   0x1ed4        ;比较结果不相等跳转到地址 0x1ed4
```

```
133    //------loop mode 9
134    vTemp=0;
135    i=100;
136    do
137    {
138      vTemp=vTemp+i;
139      i--;
140    }while(i);
141
142    //------end flag---
143    vTemp=0;
144  }
145
146
147  可以去掉此句汇编指令
148  void Main(void)
```

```
0x00001ec4  1AFFFFFA    BNE  0x1eb4
Main.c:134  :  vTemp=0;
0x00001ec8  E3A01000    MOV  R1,#0
Main.c:135  :  i=100;
0x00001ecc  E3A00064    MOV  R0,#0x64
Main.c:137  :  {
0x00001ed0  E1A00000    NOP
Main.c:138  :    vTemp=vTemp+i;
0x00001ed4  E0811000    ADD  R1,R1,R0
Main.c:139  :    i--;
0x00001ed8  E2400001    SUB  R0,R0,#0x1
Main.c:140  :  }while(i);
0x00001edc  E3500000    CMP  R0,#0
0x00001ee0  1AFFFFFB    BNE  0x1ed4
Main.c:143  :  vTemp=0;
0x00001ee4  E3A01000    MOV  R1,#0
```

图 4.3.-1　ARM 汇编结果示意图三

对于方式五代码已经没有可以更精简的空间，而方式九我们可以看出循环部分的第三条汇编指令不是必须的，因为前一条代码是执行 i-1 操作，当相减的结果等于 0 的时候会将 CPU 的状态寄存器里面的 Z 状态标志位置上，表示当前相减结果等于 0，而 BNE 指令是通过对 Z 状态标志位来进行判断，当 Z 状态标志位被置位就跳转到后面所带的地址，否则执行下一条指令，这样我们将第三条汇编指令去掉后还可以同样得到正确的程序执行结果。

那编译器能不能去掉第三条汇编指令呢？答案是肯定的，这里我暂时先不展示编译器可以去掉它的结果，因为用我现在的编译器要得到这样的结果需要使用到 C 语言编译的优化功能，下一节我会专门对优化进行讲述，到时再为大家展示。

可以不要第三条指令的原因是利用了单片机指令系统的一个规律，当进行 CPU 进行运算处理后会将状态寄存器里面 Z/C/DC/S 等标志位置位或清零来表示进位、借位、等零状态，条件跳转指令

都是依据这些状态位来决定是否进行跳转。

进行循环控制如果没有特殊的循环控制指令（某些单片机会提供循环控制指令，只要设定循环寄存器就可以让指定的代码循环执行想要的次数）就需要用一个变量进行加减来计算循环次数，通常在C语言里面会用i++/ i—方式来实现。

如果是 i++方式，循环一次都要和需要循环的次数进行比较，然后依据比较结果才知道是否还需要继续循环，但采用 i—则不一样，当达到想要的循环次数时i 自减的结果为零，这时可以直接利用状态寄存器里等零成立这一条件，选用合适的条件跳转指令来实现跳转，比如BNE/BEQ就是通过判断状态寄存器里等零是否成立来决定是否跳转。

采用i++方式也并不是不能达到与i—同样样的效果。

```
signed long i=-100;
do
{
  i++;
}while(i);   //这里一样可以在i 加到0的时候退出循环
```

但这种负数往上加的方式和日常习惯不一致，会让人们觉得别扭，所以在进行循环控制的时候，建议多用i—的方式，会在符合人们日常习惯的同时得到更高的代码效率，请记住 i—比i++效率高这一结论。

但从程序可靠性方面来说i—会不如i++。

```
i=100;
while(i)  //当i 不为0 则继续循环
{
  i—;
}
while(i<100)  //当i 小于100则继续循环
{
  i++;
}
```

如果在循环过程中i 因为意外原因导致内容被改变，比如循环了一半时i 被意外改成了10000，i—方式还要循环10000次才能退出循环，而i++方式即刻退出循环，这样i++和与i--相比i++能早一些从错误中返回。

4.4. 优化的方法与效果

用C 编程很方便的一点是代码容易模块化、循环和跳转之类的操作很好实现，免去了汇编编程那样需要用许多标号来标示程序结构以实现循环或跳转。C 语言采用大括号{}将代码进行分块，除

了可以用大括号来标示循环、选择等块外函数也是用大括号前后包括成一个整体。C 语言将变量分为全局变量和局部变量两类，局部变量的作用域被限定在一个函数之内，全局变量的作用域要大，可以在不同函数中使用。

和汇编相比，C 程序直观易懂，但这个直观易懂建立在 C 语言语法规则基础之上，会受到语法的制约，用 C 编程也会要求程序员尽量让程序结构和层次分明，这样才能更好的体现 C 语言的优点。C 的这个特性存在一个不足，就是前面的要求会让 C 代码效率和空间方面的性能下降，因为用 C 编写程序需要转换成与 MCU 指令对应汇编代码才能被解释执行，转换的过程是按照一定规则进行的，所以转换出来的代码不一定能达到直接用汇编代码编写那么简洁，加上 C 代码对程序结构和层次要求间隔清晰，在编程时容易产生一些冗余代码。

看下面一段 C 代码，对于代码作用一目了然的位置可以不添加注释。

```
行 01：void MCU_CTestFunc(void)
行 02：{
行 03：   unsigned char vAdcX,vAdcY;
行 04：   ...
行 05：   vAdcX=getAdcX();//将 ADC 得到的电压值存入中间变量
行 06：   if(vAdcX>100)
行 07：     printf("X_Voltage>100");
行 08：   else if(vAdcX>50)
行 09：     printf("X_Voltage>50");
行 10：   else
行 11：     printf("X_Voltage<=50");
行 12：   vAdcY=getAdcY();//将 ADC 得到的电压值存入中间变量
行 13：   if(vAdcY>100)
行 14：     printf("Y_Voltage>100");
行 15：   else if(vAdcY>50)
行 16：     printf("Y_Voltage>50");
行 17：   else
行 18：     printf("Y_Voltage<=50");
行 19：   ...
行 20：}
```

行 05 和行 12 先将 ADC 转换得到电压值存入中间变量，以免在行 06、08、13、15 位置再次进行 ADC 转换操作，可以减少代码执行时间；行 05 和行 12 分别用 vAdcX 和 vAdcY 来保存电压值，目的是让程序结构清晰，使 X 和 Y 两部分代码更为直观。如果我们将代码中 vAdcX 和 vAdcY 合并为 vADC，结果并不影响对程序的功能，还能节省出一个 RAM 空间，vAdcX 和 vAdcY 在代码空间上就产生了冗余。

在用汇编进行编程时，程序要求会发生变化，因为汇编语言非常不直观，需要在代码里面尽可能多的加入注释，加上使用汇编编程时程序员会重视代码的空间和效率，从而形成 C 编程中 vAdcX 和 vAdcY 代码空间冗余的几率会降低。

```
行01：   vADC  EQU 0x40   ;RAM 0x40 用来存放ADC 转换得到的电压值
行02：   ...
行03：getADCX:            ;对X 电压进行ADC 转换函数
行04：   ...
行05：   LDA ADC_REG      ;将ADC 转换结果读入累加器
行06：   STA vADC         ;将累加器中的值存入vADC
行07：   RET              ;函数返回，此时vADC 存放ADC 转换结果
行08：   ...
行09：MCU_ASMTestFunc:
行10：   ...
行11：   CALL getADCX     ;调用X 电压转换函数，执行完vADC 为X 电压
行12：   ...              ;其它代码显示X 电压区域
行13：   CALL getADCY     ;调用Y 电压转换函数，执行完vADC 为Y 电压
行14：   ...              ;其它代码显示Y 电压区域
行15：   ...
行16：   RET
```

汇编代码与 C 代码相比可以看出如果汇编不加上注释会很难看懂代码的意思，因为这些注释的存在可以不用象 C 代码那样用 vAdcX 和 vAdcY 两个变量名，只用 vADC 就可以依靠注释让代码段清功能明晰。

我们可以将 C 代码中的 vAdcX 和 vAdcY 合并为 vADC，但这样做需要 C 程序员改变约定俗成的程序风格，实际上我们不改变 C 代码也同样能实现 vAdcX 和 vAdcY 合并，编译器通过对程序的扫描可以知道在函数 MCU_CTestFunc()内 vAdcX 和 vAdcY 只用来临时存放电压值，而且两个电压值不会同时出现，这样就可以在编译的时候给 vAdcX 和 vAdcY 分配同一个 RAM，程序前半段给 vAdcX 用，后半段给 vAdcY 用，这种做法就是编译器的优化。

编译器对 C 代码编译最简单的方法就是将每一条 C 代码都转换成相应汇编指令，这种处理方法得到的汇编指令从结构层次上和原始 C 代码基本上一致，只要 RAM 等资源够用，就能比较容易做到编译出的汇编代码功能和 C 代码一样。优化则是编译器依据自己的经验，将 C 代码编译转换成汇编指令时在保证代码执行结果正确的前提下做出一些特殊调整来改良代码空间和效率。

……（详见完整版）

4.5. 全局变量的风险

不少单片机程序员喜欢用全局变量传递控制信息，比如中断标志什么的，这样做的优点是简单方便，程序员可以灵活的设定自己想传递的控制信息内容。用全局变量传递控制信息不能随心所欲，需要遵循一些规则，最常见的是不要让多方同时进行写操作的可能性存在，比如一个全局变量 x，在主程序和中断程序中都会对其进行写操作，这是风险系数非常高的操作，因为这样有可能刚好在主程序在更改 x 内容的时候中断产生，中断程序又对 x 写入另外的内容，从而导致程序判断出错，这一点只要稍有经验的工程师都会知道，但实际开发中工程师有可能知道这种风险但不能很好的避免其产生，接下来我会通过两个例子来介绍两种常见的风险疏忽。

变量宽度与单片机位宽不一致时中断使用全局变量导致出错。假定现在有一款 8bits 的单片机，我们用一个周期为 1ms 的 timer 中断做为系统时钟，timer 每中断一次会将一个 32bits 的变量加一，程序通过这个变量知道单片机上电后已经运行的时间。例子采用 C 语言编程，单片机指令系统为 6502 汇编指令。

C 代码

```
unsigned long last_time;
unsigned long ms_counter;
unsigned long get_msCounter(void)
{
  return ms_counter;
}
void main(void)
{
  ...
  ms_counter=0;
  init_timer0_irq();
  enable_irq();
  while(1)
  {
    last_time=get_msCounter();
    ...
    while(get_msCounter()<(last_time+100));
  }
}
void timer0_irq(void)
```

```
{
  CLR_TIMER0_INT_FLAG;
  ms_counter++;
}
```

C 代码汇编指令执行示意

```
unsigned long last_time;
//汇编指令用来存储变量 last_time 的 4 字节 RAM 变量
//DB  LAST_TIME_BYTE1
//DB  LAST_TIME_BYTE2
//DB  LAST_TIME_BYTE3
//DB  LAST_TIME_BYTE4
unsigned long ms_counter;
//汇编指令用来存储变量 ms_counter 的 4 字节 RAM 变量
//DB  MS_COUNTER_BYTE1
//DB  MS_COUNTER_BYTE2
//DB  MS_COUNTER_BYTE3
//DB  MS_COUNTER_BYTE4
//汇编指令用来存储函数返回参数的 4 字节 RAM 变量
//DB  FUNC_RETURN_BYTE1
//DB  FUNC_RETURN_BYTE2
//DB  FUNC_RETURN_BYTE3
//DB  FUNC_RETURN_BYTE4
//汇编指令用于计算处理的 4 字节 RAM 变量
//DB  CALC_TEMP_BYTE1
//DB  CALC_TEMP_BYTE2
//DB  CALC_TEMP_BYTE3
//DB  CALC_TEMP_BYTE4
unsigned long get_msCounter(void)
{
  return ms_counter;
//将 ms_counter 的内容存放到函数返回参数的 4 字节 RAM 变量中
  //LDA  MS_COUNTER_BYTE1          ;A=MS_COUNTER_BYTE1
  //STA  FUNC_RETURN_BYTE1         ;FUNC_RETURN_BYTE1=A
  //LDA  MS_COUNTER_BYTE2          ;A=MS_COUNTER_BYTE2
```

```
    //STA  FUNC_RETURN_BYTE2          ;FUNC_RETURN_BYTE2=A————————————③
    //LDA  MS_COUNTER_BYTE3           ;A=MS_COUNTER_BYTE3
    //STA  FUNC_RETURN_BYTE3          ;FUNC_RETURN_BYTE3=A
    //LDA  MS_COUNTER_BYTE4           ;A=MS_COUNTER_BYTE4
    //STA  FUNC_RETURN_BYTE4          ;FUNC_RETURN_BYTE4=A
    //RTS                             ;函数返回
}
void main(void)
{
    ...
    ms_counter=0;
    //将 ms_counter 清零
    //LDA  #0                         ;A=0
    //STA  MS_COUNTER_BYTE1           ;MS_COUNTER_BYTE1=A
    //LDA  #0                         ;A=0
    //STA  MS_COUNTER_BYTE2           ;MS_COUNTER_BYTE2=A
    //LDA  #0                         ;A=0
    //STA  MS_COUNTER_BYTE3           ;MS_COUNTER_BYTE3=A
    //LDA  #0                         ;A=0
    //STA  MS_COUNTER_BYTE4           ;MS_COUNTER_BYTE4=A
    init_timer0_irq();
    //对 timer0 中断进行设置，不示意汇编指令
    enable_irq();
    //开中断，不示意汇编指令
    while(1)
    //ADDRESS_1 为 while(1)死循环地址
    //ADDRESS_1:                      ;地址 1
    {
        last_time=get_msCounter();————————————————————————————————①
        //得到当前循环的起始时间
        //JSR  get_msCounter          ;调用函数 get_msCounter
        //LDA  FUNC_RETURN_BYTE1      ;A=FUNC_RETURN_BYTE1
        //STA  LAST_TIME_BYTE1        ;LAST_TIME_BYTE1=A
        //LDA  FUNC_RETURN_BYTE2      ;A=FUNC_RETURN_BYTE2
        //STA  LAST_TIME_BYTE2        ;LAST_TIME_BYTE2=A
```

```
//LDA  FUNC_RETURN_BYTE3      ;A=FUNC_RETURN_BYTE3
//STA  LAST_TIME_BYTE3        ;LAST_TIME_BYTE3=A
//LDA  FUNC_RETURN_BYTE4      ;A=FUNC_RETURN_BYTE4
//STA  LAST_TIME_BYTE4        ;LAST_TIME_BYTE4=A
...
while(get_msCounter()<(last_time+100));----------------------②
//如果循环时间达到 100ms 则开始下一次循环，未考虑溢出问题
//ADDRESS_2:                  ;地址 2
//JSR  get_msCounter          ;调用函数 get_msCounter
//CLC                         ;清 C 进位标志位
//LDA  LAST_TIME_BYTE1        ;A=LAST_TIME_BYTE1
//ADC  #100                   ;A=A+100+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE1        ;LAST_TIME_BYTE1=A，不影响 C 状态
//LDA  LAST_TIME_BYTE2        ;LAST_TIME_BYTE2，不影响 C 状态
//ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE2        ;LAST_TIME_BYTE2=A，不影响 C 状态
//LDA  LAST_TIME_BYTE3        ;LAST_TIME_BYTE3，不影响 C 状态
//ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE3        ;LAST_TIME_BYTE3=A，不影响 C 状态
//LDA  LAST_TIME_BYTE4        ;LAST_TIME_BYTE4，不影响 C 状态
//ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE4        ;LAST_TIME_BYTE4=A，忽略溢出情况
//LDA  FUNC_RETURN_BYTE4      ;A=FUNC_RETURN_BYTE4
//CMP  LAST_TIME_BYTE4        ;比较 A 和 LAST_TIME_BYTE4
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE4 跳转到地址 3
//LDA  FUNC_RETURN_BYTE3      ;A=FUNC_RETURN_BYTE3
//CMP  LAST_TIME_BYTE3        ;比较 A 和 LAST_TIME_BYTE3
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE3 跳转到地址 3
//LDA  FUNC_RETURN_BYTE2      ;A=FUNC_RETURN_BYTE2
//CMP  LAST_TIME_BYTE2        ;比较 A 和 LAST_TIME_BYTE2
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE2 跳转到地址 3
//LDA  FUNC_RETURN_BYTE1      ;A=FUNC_RETURN_BYTE1
//CMP  LAST_TIME_BYTE1        ;比较 A 和 LAST_TIME_BYTE1
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE1 跳转到地址 3
//JMP  ADDRESS_2              ;跳转到地址 2
```

```
    //ADDRESS_3:                  ;地址 3
    //JMP  ADDRESS_1              ;跳转到地址 1
  }
//RTS                             ;函数返回
}
void timer0_irq(void)
{
  CLR_TIMER0_INT_FLAG;
  //清中断标志位操作，不示意汇编指令
  ms_counter++;
  //将 ms_counter 加一
  //CLC                           ;清 C 进位标志位
  //LDA  MS_COUNTER_BYTE1         ;A=MS_COUNTER_BYTE1
  //ADC  #1                       ;A=A+1+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE1         ;MS_COUNTER_BYTE1=A，不影响 C 状态
  //LDA  MS_COUNTER_BYTE2         ;A=MS_COUNTER_BYTE2，不影响 C 状态
  //ADC  #0                       ;A=A+0+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE2         ;MS_COUNTER_BYTE2=A，不影响 C 状态
  //LDA  MS_COUNTER_BYTE3         ;A=MS_COUNTER_BYTE3，不影响 C 状态
  //ADC  #0                       ;A=A+0+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE3         ;MS_COUNTER_BYTE3=A，不影响 C 状态
  //LDA  MS_COUNTER_BYTE4         ;A=MS_COUNTER_BYTE4，不影响 C 状态
  //ADC  #0                       ;A=A+0+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE4         ;MS_COUNTER_BYTE4=A，忽略溢出情况
  //RTI                           ;中断函数返回
}
```

●注：汇编代码和 C 代码只是示意用，不能直接理解成实际情况

来看看前面代码什么情况下会出错，在位置①和位置②，程序都有调用函数 get_msCounter()，如果在位置①调用函数时 ms_counter 的值为 0x0000FFFF，当函数 get_msCounter()运行到汇编指令的位置③的时候 timer 中断产生，ms_counter 的值加一变为 0x00010000，正确的结果应该是调用函数 get_msCounter()后 last_time 的值为 0x0000FFFF 或 0x00010000，但现在结果变成了另外的值 0x0001FFFF。

程序运行到位置①时：

MS_COUNTER_BYTE1=0xFF

MS_COUNTER_BYTE2=0xFF

MS_COUNTER_BYTE3=0x00

MS_COUNTER_BYTE4=0x00

程序运行到位置③时：

LAST_TIME_BYTE1=MS_COUNTER_BYTE1=0xFF

LAST_TIME_BYTE2=MS_COUNTER_BYTE2=0xFF

MS_COUNTER_BYTE3=0x00

MS_COUNTER_BYTE4=0x00

LAST_TIME_BYTE3 和 LAST_TIME_BYTE4 还未取得新的值

此时 timer 中断产生，转去执行中断程序：

MS_COUNTER_BYTE1=0x00

MS_COUNTER_BYTE2=0x00

MS_COUNTER_BYTE3=0x01

MS_COUNTER_BYTE4=0x00

中断返回后继续取 LAST_TIME_BYTE3 和 LAST_TIME_BYTE4 的新值：

LAST_TIME_BYTE1 保持 0xFF 不变

LAST_TIME_BYTE2 保持 0xFF 不变

LAST_TIME_BYTE3=MS_COUNTER_BYTE3=0x01

LAST_TIME_BYTE4=MS_COUNTER_BYTE4=0x00

于是 last_time 得到错误的值 0x0001FFFF。

当程序运行到位置②进行循环判断时，循环控制出错，原本 100ms 的循环周期变成需要等待 65 秒才能继续下一次循环。

要避免此种风险的发生是变量位宽改为与单片机位宽一致，或者将中断修改的变量读回来后进行一次重复比较，满足前后读回的值一样的才当作有效值。

修改后的代码

```
unsigned long last_time,current_time;
unsigned long ms_counter;
unsigned long get_msCounter(void)
{
  return ms_counter;
}
void main(void)
{
  ...
  ms_counter=0;
  init_timer0_irq();
```

```
  enable_irq();
  while(1)
  {
    do                    //连续两次读得的值一样才有效
    {
      last_time=get_msCounter();
    }while(last_time!=get_msCounter());
    ...
    do
    {
      do                  //连续两次读得的值一样才有效
      {
        current_time=get_msCounter();
      }while(current_time!=get_msCounter());
    }while(current_time<(last_time+100));
  }
}
```

再来看另外一种情况，单片机依然为 8bits，中断和主程序会修改同一个变量，于是程序通过一个 8bits 变量保护标志位来对主程序的操作进行保护，当中断或主程序需要修改变量的时候，先查看变量保护标志位，如果不为 0 暂不进行修改，如果为 0 则将变量保护标志位设为非 0 值，然后进行修改变量操作，修改完毕将变量保护标志位清 0 以允许主程序或中断继续修改变量。假定 C 语言编程，单片机指令系统为 6502 汇编指令。

……（详见完整版）

4.6. 变量类型与代码效率

毋庸置疑，单片机处理和其宽度相等的数据效率最高，比如 16bits 的单片机去处理 32bits 位宽数据需要 32bits 数据分成 16bits 高低两部分后才能处理，效率自然要低不少。

采用 ARM 内核的单片机会有点特殊，ARM 内核本身是 32bits，但其支持 STRB/LDRB 这样的特殊指令，该指令可以直接读写 8bits 的数据，所以 ARM 处理 8bits 数据的时候效率不一定要比 32bits 低，但同样去传递一块数据，显然还是用 32bits 的要快，只是需要在数据块边缘用 8bits 数据进行对齐处理。

既然处理数据的效率和单片机位宽有关，那么我们在用C语言对单片机编程时就需要考虑单片机位宽，只有这样才能保证程序具有良好的效率，假设现在有两个单片机，一个为 8bits，另外一个为32bits，需要我们用这两款单片机编写一段程序，将一段数据复制到其它位置。

程序一

```
void Copy_TestFunc(char * desBuf,char * srcBuf,unsigned long size)
{
  while(size)
  {
    *desBuf=*srcBuf;          //复制1字节
    size--;                   //计数器减一
    desBuf++;                 //目的地址加一
    srcBuf++;                 //源地址加一
  }
}
```

程序二

```
void Copy_TestFunc(char * desBuf,char * srcBuf,unsigned long size)
{
  long *p1,*p2;               //用于32bits传送
  short *p3,*p4;              //用于16bits传送
  char *p5,*p6;               //用于8bits传送
  if((((long)desBuf&0x3)==0)&&(((long)srcBuf&0x3)==0))
  {
    //32bits mode             //目的地址和源地址都满足4字节对齐
    p1=(long *)desBuf;        //得到目的地址long指针
    p2=(long *)srcBuf;        //得到源地址long指针
    while(size>=4)            //还有不少于4字节的数据需要传送就循环
    {
      *p1=*p2;                //源地址传送4字节到目的地址
      size-=4;                //计数器减4
      p1++;                   //目的地址long指针自加，实际上是加了4
      p2++;                   //源地址long指针自加，实际上是加了4
    }
    p5=(char *)p1;            //目的地址改用char指针
    p6=(char *)p2;            //源地址改用char指针
    while(size)               //每次循环复制1字节到结束
```

```
    {
      *p5=*p6;
      size--;
      p5++;
      p6++;
    }
  }
  else if((((long)desBuf&0x1)==0)&&(((long)srcBuf&0x1)==0))
  {
    //16bits mode             //目的地址和源地址都满足2字节对齐
    p3=(short *)desBuf;       //得到目的地址short 指针
    p4=(short *)srcBuf;       //得到源地址short 指针
    while(size>=2)            //还有不少于2字节的数据需要传送就循环
    {
      *p3=*p4;                //源地址传送2字节到目的地址
      size-=2;                //计数器减2
      p1++;                   //目的地址long 指针自加，实际上是加了4
      p2++;                   //源地址long 指针自加，实际上是加了 4
    }
    if(size)
    {
      (char *)p3=(char *)p4;//此时只剩余1字节需要复制
    }
  }
  else
  {
    //8bits mode              //目的地址或源地址不满足2字节对齐
    while(size)               //每次循环复制1字节到结束
    {
      *desBuf=*srcBuf;
      size--;
      desBuf++;
      srcBuf++;
    }
  }
```

}

程序二与程序一相比程序结构要复杂一些，会根据源地址和目的地址的状态自动选择传送方式。对于 8bits 的单片机，每次只能传一个 8bits 的数据，所以无论程序一或者程序二单片机都会一个字节一个字节的传送，显然程序一的速度要快。对于 32bits 的单片机结果会大不一样，如果只是传送几个字节，程序二和程序一比并没有什么优势，甚至会更慢，但当传送的数据个数多而且起始地址满足 4 字节对齐时，程序二一条指令就可以传送 4 个字节，传送的速度几乎可以达到程序一的 4 倍。

所以在用 C 语言对单片机编程时，同样的 C 代码用到不同的单片机上效率可能大相径庭，我们需要结合单片机的硬件特性进行编程才能得到高效代码。标准 C 提供了不少库函数，这些库函数功能完善而且高效，使用它可以让编程人员方便不少，但要留意这些库函数有可能已经基于 32bits 平台做出了类似前面程序二的特殊处理，在非 32bits 的单片机上很有可能不能体现出高效的特性。

4.7. 慎用 int

现在用 C 编写的程序是不计其数，提供软硬件服务的厂商、具有雷锋精神的程序爱好者写出了各种各样的 C 程序代码，以便其他人在他们所提供 C 代码的基础上更快的完成产品开发，纵观这些 C 代码，你会发现里面大量使用 int 进行数据定义，实际上这不是一个好的做法。

早期的 C 语言有一个定义上的缺陷，对 int 的描述不够严谨，只是定义其为整形变量，单纯从语法上说这个定义不存在什么问题，可是 C 语言需要在相应的硬件平台上运行才有实际意义，而 C 语言没有进一步明确不同硬件平台下 int 位宽由硬件平台决定这一点，从而导致后面对 int 出现多种不同的理解。

不同的硬件平台所支持的数据位宽可能会不一样，在计算机技术发展过程中，主流的 CPU 经历了 8bits、16bits、32bits、64bits 这样的转变，不同的人在理解 int 时往往会基于当时流行的 CPU 加入自己的个人理解，我记得很清楚国内一很知名的 C 语言教材早期版本里面对 int 解释就是 16bits，正是这样教材的疏漏使得更多人将 int 理解成 16bits，后来 32bits 处理器的流行，大家又错误地默认 int 为 32bits。

对 int 的这种理解不对，实际上 int 并没有一个具体的位宽限制，由所用硬件平台（MCU）和编译器共同决定位宽为多少，通常情况下编译器会将 int 的位宽定为与所用 MCU 的位宽一致，这样 MCU 对定义为 int 的数据进行处理时因为刚好是整数不用取舍或拼凑而最为方便。

不同的 MCU 和所提供的编译器对 int 的解释各不相同，8bits、16bits、32bits、64bits 都有可能，如果在程序中使用 int 来定义数据类型，就会将一些不确定因素引入程序中，使得程序的风险系数增高。

比如编程人员忽视了编译器对 int 定义的位宽，习惯性的将 int 理解成现在大家所默认的 32bits，如果现在我们用一个 MCU 在 C 语言下开发了一款产品，所用的 MCU 和编译器为 16bits 宽度，产品需要显示开机后的时间，显示精度需要达到秒，程序员在程序里变量 SecondCounter 来纪

录开机时间，程序员将 unsigned int 错误的理解成了 32bits，认为用 unsigned int 定义的 SecondCounter 可以保证所记录的开机时间超过 100 年，完全满足任何用户的需求。

然而编译器实际是按 16bits 来处理，只能记录到 65535/3600≈18 小时。这个产品大多数应用都只是在正常上班时段，早上上班时开机，晚上下班时关机，内部测试和用户试用都是按早开机晚关机器模式进行，运行结果一切正常，于是开始量产。然而另外客户使用中发现了问题，该客户不是早开机晚关机模式，而是三班倒，需要一周甚至一个月才关机一次，产品根本无法满足客户的需求，结果是只能将已经生产的机器当作问题机处理。

这种情况主要还是因为开发人员的疏忽导致，另外一种情况就和人为疏忽无关，用 C 语言编程一大优点就是移植容易，可以很快从一个硬件平台转换到另外一个硬件平台上。产品最初阶段是采用 32bits 的 MCU，编译器解释 int 为 32bits，程序员理解没有错误也是 32bits，产品程序成功完成，运行一切正常。32bits 的 MCU 价格比较贵，成本上和其它同类产品不具竞争力，现发现有一款 16bits 的 MCU 可以满足性能需求，而且价格要低不少，于是希望改用这款 16bits 的 MCU。

先前已经写好的程序由于使用了大量的 int，直接移植到 16bits 的 MCU 编译器就将原来解释为 32bits 的 int 改解释为 16bits，显然存在数据溢出的风险，就需要我们对原程序中所有 int 都进行修订。如果我们将原来的 int 全部替换为 long 就可以避免数据溢出的风险，但这样做需要我们将程序里面所有的 int 定义逐个找出来进行替换，有一个遗漏都不算成功移植，虽然我们可以用编辑工具提供的替换功能，但替换功能会将所有“int”字符都替换掉，会将本不需要更改的地方改错。如果我们一开始就将 int 用 long 来定义，移植过程根本不需要担心数据类型解释不对的问题，使得移植程序的工作量会大为减少。

正是这些因素的存在，我的建议是尽量少在程序中使用 int 进行数据定义，C 语言提供更为精准的数据类别 char、short、long 、long long 分别严格对应 8bits、16bits、32bits、64bits，任何编译器编译出来的结果都一定满足这种对应关系，这样定义出来的数据宽度和硬件无关，在任何硬件平台上编译出来的宽度都是一样的，所以用它们定义数据类型会更好。

请记住：用 C 编写程序的时候，一定要谨慎使用 int，最好是不用。

4.8. 危险的指针

刚接触 C 语言的人可能会不大明白指针到底是什么，从我的个人理解，指针就是存储器的地址，但不能完全的等同地址，指针除了直接提供地址信息外，另外指针类型还间接告诉大家所指向的地址包含多少内容。

用一个例子来帮助大家理解指针，假设现在你是一个统领部队的指挥长，将手下的士兵成一列纵队站好，第一个士兵编号为 0，第二个编号为 1，依次类推，这里编号你就可以对应为存储器的地址。为了便于管理，你将士兵用班排连的组织来进行管理，一个班有十个士兵，一个排有三个班，一个连有三个排和另外一个独立的炊事班。

一连一排一班：士兵编号 0~9

一连一排二班：士兵编号 10~19

一连一排三班：士兵编号 20~29

一连二排一班：士兵编号 30~39

一连二排二班：士兵编号 40~49

一连二排三班：士兵编号 50~59

一连三排一班：士兵编号 60~69

一连三排二班：士兵编号 70~79

一连三排三班：士兵编号 80~89

一连炊事班：士兵编号 90~99

一连一排一班：士兵编号 100~109

二连一排二班：士兵编号 110~119

二连一排三班：士兵编号 120~129

二连二排一班：士兵编号 130~139

……

这里的连排班以及士兵编号就可以理解成指针，每个士兵是最基本的组成单位，其对应的编号对应为存储器的地址，一连二排二班表示编号 40~49 的士兵，他们现在位于队伍中的第 40+1 个人的位置，一共十个人。现在需要一个班的士兵去执行任务，你就可以通过班号选择执行任务的班，下次需要一个排你又可以通过排号来选则执行任务的排，这种方式就类似指针的操作。

前面例子也许还不能很好的解释指针，接下来直接回到 C 语言中对指针进行讲解，先从最基本的 char/short/long 类型说起。(little endian 模式)

```
char   *p_char;     //定义一个类型为 char 的指针
short   *p_short;  //定义一个类型为 short 的指针
long   *p_long;     //定义一个类型为 long 的指针
```

在 C 语言里用*号来表示指针，这一点是当时定义 C 语言语法的人决定的，没有为什么可言，当初定义语法的人选用了*号而已，如果当时选用的是其它符号现在也就是其它符号。

```
p_char=(char *)0x1000;      //将指针指向地址 0x1000
p_short =(short *)0x1000;   //将指针指向地址 0x1000
p_long =(long *)0x1000;     //将指针指向地址 0x1000
*p_char=0x12;               //地址 0x1000 内容为 0x12
*p_short=0x1234;            //地址 0x1000 内容为 0x34，地址 0x1001 内容为 0x12
*p_long=0x12345678;         //地址 0x1000 内容为 0x78，地址 0x1001 内容为 0x56，地址 0x1003 内容为 0x34，地址 0x1004 内容为 0x12
```

可以看出指向同样地址的不同类型的指针所代表的内容并不相同，char 的指针为一个字节的内容，而 short 和 long 却分别为两个字节和四个字节，显然指针除了带有地址信息外还有所包含内容大小的信息，这就是指针与地址所不同的地方。

注意将指针指向一个地址的操作并不是使用 p_char=0x1000 这样的语句直接等，而是使用了(char *)进行强制转换，如果不加(char *)的话编译会报告错误。这是C语言使用指针时的一个限制，所有对于指针的操作必须是同类型的指针才可以进行，对于 0x1000 这样一个数字，虽然我们可以当它是一个地址，但用做指针的时候需要表明指针类型才能使用，否则程序无法知道*0x1000这样的操作表示的到底是一个字节的char类型还是两个字节的short类型，C语言有这样的限制后就可以让指针和地址区分开。

用C语言编程变量的分配大都由编译器决定，也就是说程序员可以不用理会所用变量究竟放在什么位置，而用C语言对单片机编程许多时候需要直接对某个地址进行读写操作，引入指针就会使这类操作变的更简单直接。对于char类型的指针，每个指针只对应一个字节的内容，而short类型则是两个字节，C语言为了提升效率对于这类指针做出了起始地址对齐的要求，比如short指针地址需要能被二整除，而long指针地址则需要能被四整除。

这里不想对什么是指针这样的概念性问题阐述过多，总之一点是指针让程序控制存储器空间变的非常简捷，但正是这个简捷打破了C语言的封闭性，让程序员可以随心所欲的去读写存储器空间，所以在便捷的同时也给程序带来了一定的风险性。

现在需要用程序来完成这样的功能：从地址0x1000开始将0、1、2、…、255依次写入头256字节，为了检验程序是否写入正确还需要将写入内容回读回来进行校验。

正确的代码：

```
long i;
char *p1,*p2;          //这里指针类型是char
p1=(long *)0x1000;     //将指针p1地址指向0x1000
p2=(long *)0x1000;     //将指针p2地址指向0x1000
for(i=0;i<256;i++)
{
  *p1=i;               //将指针内容依次写为0、1、2、…、255
  p1++;                //指针加一，指针所指向地址随之加一
}
for(i=0;i<256;i++)
{
  if(*p2!=i)
  {
    while(1);          //如果校验出错则进入这个死循环
  }
  p2++;                //指针加一，指针所指向地址随之加一
}
```

错误的代码：

```
long i,*p1,*p2;         //这里指针类型是long
p1=(long *)0x1000;      //将指针p1地址指向0x1000
p2=(long *)0x1000;      //将指针p2地址指向0x1000
for(i=0;i<256;i++)
{
  *p1=i;                //将指针内容依次写为0、1、2、…、255
  p1++;                 //指针加一，实际上指针所指向地址加四
}
for(i=0;i<256;i++)
{
  if(*p2!=i)
  {
    while(1);           //如果校验出错则进入这个死循环
  }
  p2++;                 //指针加一，实际上指针所指向地址加四
}
```

错误代码的原因是未能正确使用指针，对于char类型的指针，执行自加或者加一操作后指针地址同样是加一，而short和long则不一样，分别加二和加四。

```
char *p=0x1000;    //char类型指针p指向地址0x1000
p++;               //执行完该指令后指针p指向地址0x1001，注意这里指针地址加一
long *p=0x1000;    // char类型指针p指向地址0x1000
p++;               //执行完该指令后指针p指向地址0x1004，注意这里指针地址加四
```

正确写入的结果：

地址　　内容

0x1000　0x00

0x1001　0x01

0x1002　0x02

…

0x10FF　0xFF

错误写入的结果：

地址　　内容

0x1000　0x00

0x1001　0x00　　多写入的0x00

0x1002　0x00　　多写入的0x00

0x1003　0x00　　多写入的 0x00

0x1004　0x01

0x1005　0x00　　多写入的 0x00

0x1006　0x00　　多写入的 0x00

0x1007　0x00　　多写入的 0x00

…

0x13FC　0xFF

0x13FD　0x00　　多写入的 0x00

0x13FE　0x00　　多写入的 0x00

0x13FF　0x00　　多写入的 0x00

可以看出错误是每写一个数据会另外在后面多写入三个 0x00，同时地址往后加四，虽然代码也有进行回读校验的操作，但回读操作完全是写入操作的逆向过程，所以这个校验不会发现该错误，如果不发生空间溢出错误，这个错误就会被隐藏起来而不被发现，从外表看程序运行一切正常，但内部的实际运行结果并不是程序员所预期的。

不要认为这种错误不严重，如果位于 0x1100~0x13FF 的空间有其它作用，前面的错误就会导致这部分空间的内容被错误修改，从而导致程序运行出错，严重的可以让程序崩溃。

使用指针最大的风险是容易产生空间溢出或者地址错误的情况，用 C 语言对数组或者变量进行读写的时候，虽然程序员不清楚这些数组或者变量在存储器中的具体位置，但读写操作是用数组或变量名来进行读写的，所以很容易做到所需要进行读写的对象不出错，如果在编写程序时因为书写等原因让名称出错，编译器也会报告名称错误，这样读写到错误位置的几率会相当小，除非在代码中定义了名称非常相似的数组或变量名。而用指针来进行读写则不一样，需要程序员自我对读写的位置进行保护确认，只要有一点疏漏就可能导致错误的产生。

来看一个例子，这个例子是因为其它用途方便而定义了四个 64 字节的数组，现在想将 0~255 分别写到这四个数组里面去，数组 Temp_BufA[]写入 0~63，数组 Temp_BufB[]写入 64~127，数组 Temp_BufC[]写入 128~191，数组 Temp_BufD[]写入 192~255。

```
char Temp_Byte;
char __align(256) Temp_BufE[256];  //__align(256)表示起始地址需要256字节对齐
char Temp_BufA[64];
char Temp_BufB[64];
char Temp_BufC[64];
char Temp_BufD[64];
char *p;
long i;
p=Temp_BufA;                          //指针地址指向数组Temp_BufA[64]的首字节地址
```

```
for(i=0;i<256;i++)
{
  *p++=i;                          //依次写为 0、1、2、…、255
}
```

可能不少人会认为这段程序没有问题，他们会说数组 Temp_BufA[]到数组 Temp_BufD[]是连续定义的，在程序里面所占用的存储空间自然也是连续的，前面的程序只用一个指针循环就可以将这四个数组写入所需的值，所以程序不但没有问题，而且还是一段很精简的代码，值得大家学习。实际情况并不如此，我们并不能保证数组 Temp_BufA[]到数组 Temp_BufD[]占用连续的存储空间，有可能这四个数组是被间隔开。

假定存储空间的分配是从地址 0x1000 开始，编译器看到 Temp_Byte，于是将地址 0x1000 分配给 Temp_Byte 用，可接下来的是数组 Temp_BufE[]，需要 256 字节对齐，显然不能再将 0x1001 分配为数组 Temp_BufE[]的起始地址，而是分配 0x1100 才满足要求。这时编译器处理到数组 Temp_BufA[]，因为地址 0x1001~0x10FF 区间还没有被分配出去，所以将 0x1001~0x1040 区间分配给数组 Temp_BufA[]，而不是从地址 0x1200 开始分配。

与数组 Temp_BufA[]一样数组 Temp_BufB[]和数组 Temp_BufC[]会被分配到地址 0x1041~0x1080 和 0x1081~0x10C0 这两段区域。但处理到数组 Temp_BufD[]时遇到新问题，显然数组 Temp_BufD[]需要连续的 64 字节，而现在从地址 0x10C1 开始到 0x10FF 只有 63 个字节的剩余空间，所以数组 Temp_BufD[]所分配的空间只好分配为地址 0x1200~0x123F 这一段，这样四个数组的空间并不连续，如果运行前面的代码自然就会得到错误的结果。

按前面假定编译器可以得出的存储器分配空间如下（实际情况不一定和此相同）：

地址	内容
0x1000	变量 Temp_Byte
0x1001~0x1040	数组 Temp_BufA[64]
0x1041~0x1080	数组 Temp_BufB[64]
0x1081~0x10C0	数组 Temp_BufC[64]
0x10C1~0x10C3	空余区间
0x10C4~0x10C7	指针*p
0x10C8~0x10CB	变量 i
0x10CC~0x10FF	空余区间
0x1100~0x11FF	数组 Temp_BufE[256]
0x1200~0x123F	数组 Temp_BufD[64]

指针有的时候会有地址对齐的要求，比如 short 和 long 类型的指针分别要求两字节和四字节对齐，有的编译器并不能对这些细节进行可靠的检查，稍有不慎就可能用错指针。

```
long *p;          //申明类型为 long 的指针，按要求需四字节对齐
```

p=(long *)0x1001; //这里将指针地址指向 0x1001

这样的代码有的编译器不会报告错误，而实际上类型为 long 的指针地址是必须四字节对齐的，所以代码带有错误，执行这样的代码会导致错误发生，而程序员还以为自己已经成功将地址 0x1001 交给指针 p。

指针空间溢出的错误在实际中最容易发生，常常会遇到这样的情况，变量没有被任何代码直接修改，但程序运行的结果发现变量内容被修改成程序员所不期望的内容，这种错误大都是使用指针空间溢出导致的。对于这种错误由于不能通过查看代码是否更改了变量，加上错误的产生是偶然的，需要在某些特定条件下才会出现，所以如果仿真调试工具功能不是很强大的话很难调试跟踪这种错误。

虽然这种错误经常会遇到，但形成的原因往往错综复杂，不大容易用一个简洁的例子就能说明清楚，这里给出一个例子，错误产生的根源也许并不能完全归于指针，但确实是和使用指针有关。我们通过 UART 收发数据，数据依照下面格式进行传送，最大长度不超过 1024 字节。（例子代码是针对演示错误而特意写成下面形式）

Byte1　　同步字 0xFF，用来标示数据包开始

Byte2~3　两个字节用来表示数据包长度

Byte4~N　数据内容，规定数据包中不可以包含 0xFF 以免与同步字冲突

ByteN+1　数据包内容校验码低位字节

ByteN+2　数据包内容校验码高位字节

接收端代码：

```
unsigned short receive_count;          //数据接收计数器
unsigned short check_sum;              //用来计算数据包校验码
unsigned char receive_buf[1024],*p;    //数据存放缓冲区和接收存放指针
char receive_flag=0;                   //数据接收状态标志
UART_RxInt()                           //每收到一个字节都会触发该中断函数
{
  unsigned char temp;
  temp=*UART_RX_DATA;                  //将接收到的数据读到 temp 中
  …//其它代码
  if((receive_flag==0)&&(temp==0xFF))  //如果接收状态为 0 且当前收到同步字
  {
    receive_falg=1;                    //接收状态转为 1
    p=receive_buf;                     //接收指针指向接收缓冲区首地址
  }
  else if(receive_flag==1)             //如果接收状态为 1
```

```
    {
      receive_flag=2;                       //接收状态转为 2
      receive_count=temp;                   //保存数据包长度低位字节
    }
    else if(receive_flag==2)                //如果接收状态为 2
    {
      receive_flag=3;                       //接收状态转为 3
      receive_count|=(temp<<8);             //保存数据包长度高位字节
    }
    else if(receive_flag==3)                //如果接收状态为 3
    {
      if(receive_count>0)                   //保存数据直到数据接收计数器为零
      {
        *p++=temp;                          //保存数据到缓冲区，指针地址加一
        receive_count--;                    //数据接收计数器减一
      }
      else
      {
        receive_flag=4;                     //接收状态转为 4
      }
    }
    else if(receive_flag==4)                //如果接收状态为 4
    {
      receive_flag=5;                       //接收状态转为 5
      check_sum=temp;                       //得到校验码低位字节
    }
    else if(receive_flag==5)                //如果接收状态为 5
    {
      receive_flag=0;                       //接收状态转为 0
      check_sum|=(temp<<8);                 //得到校验码高位字节
      …                                     //数据处理代码
    }
  }
```

不能说上面代码是错误的，正常情况下上面的代码运行结果都不会有什么问题，但这部分代码是用来接收另外设备通过 UART 发送过来的数据，这个收发过程可能会因为外部干扰等原因造成传

输数据出错，当这种情况发生时，程序就会变的不在稳定可靠。实际上前面的代码也有考虑到数据传输可能会出错，在数据包中加上校验码就是用来判断数据传输是否出错，如果数据传输出错，校验码会不对，这样程序就可以把所接收到的数据包做相应处理。

但对这部分代码传输出错的考虑不够周到，只是去判断数据包是否出错，并没有去考虑出错位置不同可能带来的不同影响。干扰是随机的，所以传输的错误可以产生在数据包的任意位置，如果果然刚好在传送数据包长度的时候导致数据传输出错，比如现在数据包长度原本是 0x200(512 字节)，干扰导致这个长度变为 0x1200(4608)字节，这时前面代码面临的问题就非常严重，会从 receive_buf[]的起始位置开始存放 4608 字节后才终止接收过程，而 receive_buf[]只有 1024 字节的空间，也就是说后面会有 4608-1024=3584 字节的空间会被指针错误修改。

象这种错误的调试跟踪会非常困难，因为它发生的概率本身就非常小，可以说连续发生两次的几率几乎为零，而这个错误一旦发生，很有可能让整个程序崩溃，即便当时想通过仿真调试工具去查看现场也是很困难，可以说是无法通过调试的方法手段来找出这类错误根源，如果想避免这类错误，只能是在编写代码阶段就充分考虑到各种可能性，在代码中对各种异常做出相应保护而避免意外出错。

真可谓是指针虽好，可用起来并不简单，刚开始接触单片机 C 语言编程的朋友还是多用数组来编写代码比较好一些，当自己对 C 语言有一个良好的掌握程度后再去逐步使用指针，使用指针的时候一定要仔细确认指针是不是依照自己的想法在变动，最好在指针改变后查看一下地址是否正确。

4.9. 循环延时

用 C 语言编写单片机程序时，常会遇到一些需要延时等待的情况，不少程序员为了图方便经常会用 for(i=0;i<1000;i++)这样循环来实现延时，我自己有时候也都这么做，这不是一个好的方法。

首先这样的循环代码写出来后程序员自己也不清楚这段代码到底延时有多久，还需要用仿真器观察对应汇编指令或者用仪器测试延时时间的大小。

其次所得到延时时间稳定性不够好，如果在延时循环中有中断产生就会使得延时变大，所以在实际程序运行时执行完这个循环所耗用的时间存在长短不一的可能，中断对循环延时的影响还不是特别大，因为其它方式实现延时也会受到中断的影响，比如我们想实现延时 100us，延时循环中一个需要耗时 200us 的中断产生，这样不管用什么方法，当响应完中断返回时时间就多出了 200us，也就是说这时无论什么方法来实现的延时都不会少于 200us。对循环方式延时影响真正大的是编译优化、系统时钟改变这两种操作。

我们来看一下用 ADS 编译器不同优化条件下对循环 for(i=0;i<1000;i++)的影响，先关闭优化功能，可以看出循环部分代码需要四条汇编指令

```
203     IO_init();
204     MCU_CTest();
205
206     for(i=0;i<1000;i++);
207
208 while(1)
209 {
210         *((unsigned int*)
211         *((unsigned int*)
212         *((unsigned int*)
213         *((unsigned int*)
```

```
Main.c:203   :  IO_init();
0x00001f50   EB002685   BL    0xb96c IO_init
Main.c:204   :  MCU_CTest();
0x00001f54   EBFFFF99   BL    0x1dc0 MCU_CTest
Main.c:206   :  for(i=0;i<1000;i++);
0x00001f58   E3A04000   MOV      R4,#0
Main.c:206   :  for(i=0;i<1000;i++);
0x00001f5c   E3540FFA   CMP   R4,#0x3e8
0x00001f60   AA000001   BGE   0x1f6c
Main.c:206   :  for(i=0;i<1000;i++);
0x00001f64   E2844001   ADD   R4,R4,#0x1
0x00001f68   EAFFFFFB   B     0x1f5c
Main.c:208   :while(1)          循环部分代码
0x00001f6c   E1A00000   NOP
```

图 4.9.-1　ADS 对 for 循环无优化编译结果图

再来看看打开优化功能后的情况，循环部分的代码从原来的四条变成了三条，也就是说在硬件不做任何变动的情况循环的时间会改变为原来的四分之三，这个影响是非常明显的，很有可能就形成延时不够的情况。

```
203   IO_init();
204   MCU_CTest();
205
206   for(i=0;i<1000;i++);
207
208 while(1)
209 {
210       *((unsigned int*
211       *((unsigned int*
212       *((unsigned int*
213       *((unsigned int*
214   clrSCR();
```

```
Main.c:203   :  IO_init();
0x00001e7c   EB001AD5   BL   0x89d8 IO init
Main.c:204   :  MCU_CTest();
0x00001e80   EBFFFFCB   BL   0x1db4 MCU_CTest
Main.c:206   :  for(i=0;i<1000;i++);
0x00001e84   E3A00000   MOV  R0,#0
0x00001e88   E289895C   ADD  R8,R9,#0x170000
0x00001e8c   E28D604C   ADD  R6,R13,#0x4c
Main.c:206   :  for(i=0;i<1000;i++);
0x00001e90   E2800001   ADD  R0,R0,#0x1
Main.c:206   :  for(i=0;i<1000;i++);
0x00001e94   E3500FFA   CMP  R0,#0x3e8
0x00001e98   BAFFFFFC   BLT  0x1e90
Main.c:211   :    *((unsigned int*循环部分代码
0x00001e9c   E589A024   STR  R10,[R9,#0x24]
```

图 4.9.-2　ADS 对 for 循环有优化编译结果图

对于系统时钟改变的情况很容易理解，系统时钟被改变，每条指令执行的时间同样会相应发生变化，举个极端的例子，原本是用 8MHz 的晶振现在改用 4MHz，每条执行执行的时间就会增加一倍，循环所花的时间自然也就增加了一倍。有人可能有疑问，我们在实际应用中不会改用不同频率的晶振，那不就不用担心这样的情况了吗？是的，大多数时候不用担心这个问题，但在某些特殊情况下就需要考虑该问题，比如功能强大一点的 MCU 可以通过 PLL 来设定系统时钟，即便不改变晶振频率也可以改变系统时钟，当系统时钟频率高的时候 MCU 处理速度快，但会有芯片发热大、对外辐射增加、稳定性变差这些问题产生，所以有的产品当需要高速处理数据的时候就将 PLL 设为高频率以满

足数据处理需求，当数据处理完又恢复为PLL低频率得到一个稳定可靠的工作状态，对于这种情况循环延时就会不准确。

我们知道电源电压波动、电阻阻值误差这些因素都会影响 RC 振荡器的工作频率，所以当 MCU 使用RC振荡器的时候也会让循环延时不准，PLL还可以根据对PLL的设定值判断当前工作频率来解决时钟变化导致的影响，而 RC 振荡器则完全无法知道时钟的变化，通过程序是无法让循环延时修正误差。

如果想要避免因优化对循环延时产生的影响，建议用汇编指令来延时，编译器不会对汇编指令进行优化，所以编译器优化的选择与否不会影响汇编指令的那部分代码，汇编语言写的延时代码只要系统时钟不变就能保证延时恒定。另外通过查询MCU的编程手册，可以知道每条汇编指令所需要的机器周期数，汇编语言写的延时代码可以精确一个机器周期，程序员通过计算汇编指令数就能知道延时时间的长短，从而解决了用C语言写循环代码延时时间不透明的问题。

时钟改变的情况不容易处理，只能是在程序中依据硬件的实际情况提前做出预测，尽量做到延时时间不小于最低要求，从而保证程序的可靠执行，比如现在采用的是 RC 振荡器，经过分析预计其工作频率可能存在2%的误差，如果需要延时100微秒，MCU一个机器周期是1/8微秒，我们就要保证延时代码的理论延时不小于102*8=816个机器周期，否则可能有少量硬件延时达不到100微秒，对于实际应用还需要留有足够的余量，我一般是将这个余量定为20%，也就是说按120微秒来编写代码。

这段以6502指令为基础的C汇编延时代码可以用来帮助大家了解如何在C语言编程中实现汇编代码延时，假定是指令周期等于机器周期，不考虑中断的影响。

```
void _delay_cycle(unsigned char n)
{
#ASM                  ;标示下面为嵌入到C代码中的汇编代码
  TAX                 ;执行该汇编指令需1个机器周期，C函数参数传入
DELAY_100LOOP:        ;循环体执行需要3*X-1 个机器周期
  DEX                 ;执行该汇编指令需1个机器周期
  BNE  DELAY_LOOP     ;跳转回去需2 个机器周期，结束循环需1 个机器周期
#ENDASM               ;标示嵌入到C 代码中的汇编代码结束
  //C语言函数返回需要2个机器周期
  //RTS
}
```

调用示例：

```
_delay_cycle(100);
```

```
//C函数输入参数n通过累加器A传人，将n写入累加器A需1个机器周期
//LDA  #100
//调用函数需2个机器周期
//JSR  _delay_cycle
```

可以看出函数被调用一次需要耗费 1+2+(1+(n*3-1)+2)=n*3+5 个机器周期，n 的取值范围为 0~255，这样通过调用该函数可以实现的延时为5~770个机器周期，间隔为每档3个机器周期,如果是一个系统时钟为8MHz 的系统，该函数的理论延时是 0.625~96.25 微秒，间隔为 0.375 微秒。

上面是用循环实现延时，即便是用汇编代码来实现从我个人的角度看也不值得提倡，比较好的做法应该是充分利用MCU所带的timer资源，用timer中断实现延时，这样做在计算延时时间时要便捷不少，同时还能保证到具有良好延时精准度。

这章最后用 C 的伪代码给大家示意一下如何可以通过 timer 中断实现精确延时，假定可以将 timer 中断设置到比较高的中断优先级，通过 timer 中断来停止在主程序中启动的 testing，这样的做法可以让延时非常精准。

```
void (*timer_callback)(void);       //定义一个函数指针
unsigned long timer_int_flag=0;     //定义timer中断标志变量
void timer_irq(void)                //timer中断函数
{
  if(timer_callback!=NULL)
  {
    timer_callback();               //如果函数指针不为空则执行对应函数
                                    //如指针为stop_testing则执行stop_testing（）
  }
  DISABLE_TIMER_IRQ();              //禁止timer中断，具体代码忽略
  timer_callback=NULL;              //清函数指针为空
  timer_int_flag=1;                 //置timer中断标志
}
void timer_delay(unsigned long us,void *callback)
                                    //将第二输入参数的指针给函数指针
{
  timer_callback=callback;          //将第二输入参数的指针给函数指针
  set_timer_counter(us);            //将timer相关寄存器设定为所需要的内容
                                    //具体代码忽略
  ENABLE_TIMER_IRQ();               //使能timer中断，具体代码忽略
}
```

```
void start_testing(void)          //启动测试函数
{
  ...                             //启动测试的代码，具体代码忽略
}
void stop_testing(void)           //停止测试函数
{
  ...                             //停止测试的代码，具体代码忽略
}
void main(void)
{
  ...
  timer_int_flag=0;               //清 timer 中断标志
  start_testing();                //调用启动测试函数
  timer_delay(10000,stop_testing); //延时 10000us 后产生 timer 中断停止测试
                                  //在 timer 中断中会执行函数 stop_testing()
  while(time_int_flag==0);        //等待 timer 中断标志被置为 1
  ...
}
```

例子中的伪代码没有考虑代码自身产生的延时，也没有考虑如果有优化时对全局变量应采取的相应保护措施。留一个问题给大家，为什么需要将停止测试的函数 stop_testing()放在 timer 中断中，而不是在 while(time_int_flag==0);之后调用？

4.10. 运算表达式

用 C 语言编程，少不了使用运算表达式，C 语言的运算表达式和日常生活中的数学表达式非常接近，所以运算表达式对使用者来说非常直观，也很容易理解。数学里面一个数是正是负、是整数还是小数可以用负号和小数点来让人知道这个数的类型，知道了数的类型人们就可以按照数学法则进行运算。但 C 语言毕竟不是我们上小学就开始学习的数学，因为计算机技术的限制，在 C 程序中出现整数和小数、有符号数和无符号数混用等情况时，为了保证计算结果正确，C 语言对这些情况做出了一些自己的特性约定，让不同类型的数混用时按某一个规律自动进行转换，这样一来就出现同样的运算表达式 C 语言和日常生活中的数学运算结果不相同的情况。

在用 C 语言进行编程时，一定要留意它和日常生活中数学在处理运算表达式方面的不同，否则就会让程序出现一些本可以避免的错误。

我曾经遇到过这样的情况，用 check_sum 对通讯的数据进行一个简单的保护，保护的方法相当

简单，就是将通讯的数据累加求和，发送完数据后再将计算所得的和发送给接收方，接收方也同样将所接收的数据累加求和，然后和发送过来的校验和做比较，如果两者一致就认为数据传输过程是可靠的。按说这个方法很简单，不大可能出错，可实际情况就是我所写的 C 程序会出错，会将原本传输正确的数据错误判断为传输出错。

出错的原因就是对 C 语言的运算表达式理解有误，C 语言在进行处理运算表达式的时候有一条基本法则：如果在运算表达式中有不同类型的变量，运算的时候是 signed 遇到 unsigned 会自动将 signed 转换成 unsigned，数据位宽度小的自动转换成和大的数据位宽一致。

暂先不对这条基本法则做过多的解释，看看我当时是如何出错的，接收方 C 代码是将接收到的数据放到 unsigned char data_buf[]中，再将全部数据累加求和得出校验码。发送方采用汇编编写代码，这里不列出具体代码。

```
unsigned char buf[7];               //前面 6 个字节放数据，第 7 个字节为校验码
unsigned char receive_flag;         //用来表示校验结果的标志变量
if((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5])==buf[6])
{
  receive_flag=DATA_CHECK_RIGHT;  //校验结果正确，认为传输可靠
}
else
{
  receive_flag=DATA_CHECK_ERROR;  //校验出错，认为传输有错误
}
```

传输的数据包长度不长，为节省代码，没有用循环语句而是直接将 6 个字节的数据相加求和，当时我的理解是 if((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5])==buf[6])表达式中全部是同一类型的 unsigned char 数组成员变量，那计算过程也应该是自动按 unsigned char 类型处理，如果数据相加求和溢出也会将求和自动转换成 unsigned char 类型，比如这 6 个字节加起来的和为 0x0123,会自动将这个和处理成 0x23，溢出的高位将被舍弃掉，这样代码是可靠的。

实际情况并不是我所想像的样子，假如现在 buf[7]={0x11,0x22,0x33,0x44,0x55,0x66,0x65}，人工计算 0x11+0x22+…+0x66=0x165，那么汇编计算出来的校验码应是 0x65，和数组最后一个字节一致，表明这次传输的数据是可靠的，但 C 代码执行的结果并不是这样，会报告数据校验出错，通过调试查看 C 代码对应的汇编指令知道错误原因，对数据相加的结果不是和我所想的一样会自动转换成 unsigned char 类型，这个时候是将加出来的结果按 MCU 的位宽来处理，所以校验出错，正确代码应如下所示。

将运算表达式的结果先进行强制转换。

```
if(((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5])&0xFF)==buf[6])
//或 if((unsigned char)((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5]))==buf[6])
{
```

```
  receive_flag=DATA_CHECK_RIGHT;  //校验结果正确，认为传输可靠
}
else
{
  receive_flag=DATA_CHECK_ERROR;  //校验出错，认为传输有错误
}
```

或者先将运算表达式的结果赋给指定变量。

```
unsigned check_sum;
check_sum=buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5];
if(check_sum==buf[6])
{
  receive_flag=DATA_CHECK_RIGHT;  //校验结果正确，认为传输可靠
}
else
{
  receive_flag=DATA_CHECK_ERROR;  //校验出错，认为传输有错误
}
```

之所以产生这样的错误，是因为我对 C 语言的运算表达式的处理方式理解不够透彻，对运算结果用想当然的方法去理解。接下来看看运算表达式中有不同类型的变量时 C 语言究竟是如何进行自动转换的，例子中的结果以 ADS 编译器结果为准，其它编译器可能得出不同结果。

```
signed char x;
unsigned char y;
unsigned short z;
x=-1;          //用十六进制表示为 0xFF(-1)
y=x;           //运行结果为 z=0xFF(255)
z=x;           //运行结果为 z=0xFFFF(65535)
```

当 y=x 时，y 和 x 都是 char 类型，直接将 x 的内容从有符号转换成无符号类型就赋给 y，但 z 和 z 除了无符号和有符号的区别外数据位宽也不一样，分别是 16bits 和 8bits 宽，x 自身为-1，但是 8bits 类型，当将其赋给 z 时，比 y=x 需要多数据位数的扩展，这样就要将原本 8bits 的-1(0xFF)转换成 16bits 的-1(0xFFFF)，所以最后 z 的值为 0xFFFF，和 x 的 0xFF 不同。

这里有一点要特别提醒大家：在 C 语言里不要将-1 和 0xFF 视为等同数据，大多数编译器会将 0xFF 当做 255 来处理。

```
signed char x;
```

```
x=-1;          //用十六进制表示为 0xFF(-1)
if(x==0xFF)    //实际上这句等效于 if(x==255)
{
  x=-1;        //程序不会进入此处
}
if(x==-1)
{
  x=0;         //程序会进入此处
}
```

```
Main.c:88    :  if(x==0xFF)
0x00001e14   CMP           R1,#0xff
0x00001e18   BNE           0x1e20
Main.c:90    :    z=0;  不同的比较指令
0x00001e1c   MOV           R0,#0
Main.c:92    :  if(x==-1)  CMN表示和-1比较
0x00001e20   CMN           R1,#0x1
0x00001e24   BNE           0x1e2c
Main.c:94    :    z=0;
0x00001e28   MOV           R0,#0
Main.c:96    :}
0x00001e2c   BX            R14
```

图 4.10.-1　ADS 对-1 和 0xFF 编译对比结果图

接下来是一段奇妙的代码，也许会给你一种不可思议的感觉，但实际结果确实如此，如果你理解了这段代码，我想你对 C 语言运算表达式中的自动转换规则已经是非常清楚。

```
signed char x;
x=128;         //注意 signed char 有效范围为-128~127
if(x==128)
{
  x=128;       //程序不会进入此处
}
if(x==-128)
{
  x=0;         //程序会进入此处
}
```

是不是有点奇妙？明明用 C 代码给 x 的值是 128，在后面进行比较就变成了-128。如果你明白

了计算机内部的数据表示方法后就不难理解其中奥妙，我们知道计算机内部数据采用二进制表示方法，正数是多少就是对应的二进制数，负数则先将数的二进制数取反，然后加一。十六进制是让人们对二进制数可以更直观表示的中间方法，数值自身的内容完全和二进制一致。来看看十六进制是如何表示 8bits 数的：1 为 0x01，-1 为 0xFF(0x01 取反得到 0xFE 再加一得到 0xFF)。

C 语言在处理运算表达的时候，如果表达式中有数字，通常会将这个数字默认为与 MCU 位宽一致的整型数，再用这个整型数的二进制数进行运算处理，例如上面的 x=128 编译器会先将 128 转换成 0x00000080，因为是要将这个数给 signed char 类型的 x，所以在产生汇编指令的时候就只需要 0x80 作为操作数，另外三个字节在生成汇编指令阶段就给抛弃掉。

```
signed char x;
x=0x1234;
if(x==0x34)     //产生汇编指令的时候只将低位字节 0x34 给 x
{
  x=0;          //程序会进入此处
}


signed char x;
x=-129;         //十六进制为 0xFFFFFF7F
if(x==0x7F)
{
  x=0;          //程序会进入此处
}
```

到这里就更加容易理解前面的奇妙代码。

```
signed char x;
x=128;          //实际上是将十六进制数 0x00000080 的低位字节 0x80 给 x
if(x==128)      //实际上是将 x 和十六进制数 0x00000080 进行比较
                //可以理解成 if((-128)==128)，也就是 if(0xFFFFFF80==0x00000080)
{
  x=128;        //显然比较结果不相等程序不会进入此处
}
if(x==-128)     //实际上是将 x 和十六进制数 0xFFFFFF80 进行比较
                //可以理解成 if((-128)==（-128）)，也就是 if(0xFFFFFF80==0xFFFFFF80)
{
```

```
    x=0;        //比较结果相等程序会进入此处
}
```

转换规律小结：

有符号类型和无符号类型混用时自动转换为无符号类型。

数字默认为与 MCU 位宽一致的整型数值。

不同位宽的数据（或变量）混用时自动转换成宽度更宽的类型。

不同宽度的数据进行赋值运算时候自动进行高位扩展或截取。

还有一点同表达式有关，C 语言对于运算符自己有一张优先级表，不少人编写程序时可能是为了减少表达式长度或证明自己对 C 很熟悉，在代码中充分利用运算符的默认优先级。

```
if(a>b && b>0)
a=10>4&&!(100<99)||3<=5
```

不能说这种写法错误，有些经典样例代码可能都是按此方式书写，但我个人绝不赞同这种写法。即便是经典样例代码，也是基于艺高人胆大的基础之上，不值得推广，稳妥的做法是用括号将表达式的优先顺序括起来。这样做虽然代码会繁琐一些，但通用性和可读性强，就算是记不住各级运算符优先级的人也不会弄错运算的先后顺序，编译得到的汇编指令也不会变多。

```
if((a>b) && (b>0))
a=(10>4)&&(!(100<99))||(3<=5)
```

后一种写法是不是要清晰许多？如果你以前习惯采用 C 默认优先级的方式，来看一下你真的记住了所有的运算符优级顺序了吗？

优先级	运算表达式
最高	()(小括号) [](数组下标) .(结构成员) ->(指针型结构成员)
↑	!(逻辑非) .(位取反) -(负号) ++(加 1) --(减 1) &(变量地址)
↑	*(指针所指内容) type(函数说明) sizeof(长度计算)
↑	*(乘) /(除) %(取模)
↑	+(加) -(减)
↑	<<(位左移) >>(位右移)
↑	<(小于) <=(小于等于) >(大于) >=(大于等于)
↑	==(等于) !=(不等于)
↑	&(位与)
↑	^(位异或)
↑	|(位或)

↑	&&(逻辑与)
↑	||(逻辑或)
↑	?:(?表达式)
↑	= += -=(联合操作)
最低	,(逗号运算符)

图 4.10.-1　标准 C 运算符优先级表

4.11. 溢出

除了宇宙，一切事物都是有限的，水杯子满了水会溢出，水缸满了水也会溢出，水池满了水还是会溢出，就是水库满了同样也是溢出，没有其它选择。在编写单片机程序时，程序员同样要面对数据溢出情况，如果在写程序时不预先对数据溢出情况加以考虑，最终结果很可能是错误隐藏在程序之中，在特定条件下就会暴露出来导致程序出错。

实际上无论是汇编还是 C 语言，都需要面对数据溢出情况，只不过程序员在使用汇编编程时候，因为没有 C 语言中的 char/short/long 这些数据类型，需要程序员自己对数据的位宽进行处理，所以这个时候程序员往往会下意识的留意到数据溢出情况，而使用 C 语言时，数据类型的处理由 C 语言编译器完成，程序员关注的重点会放在程序结构方面，常常忽略数据类型这些细节，所以相较汇编语言使用 C 语言编程时更容易在数据溢出方面出问题。

我自己就有在数据溢出方面考虑不周差点让产品出大问题的经历，该产品是用一片小容量的 SPI FLASH 芯片来存储数据，数据存储采用自定义的文件系统格式，为了让存储的数据可靠性更高，我将文件系统的 FAT（文件分配表）的内容用 CHECK SUM 进行校验，只有校验正确的记录才会被当作有效数据。

……（详见完整版）

4.12. 强制转换

不少用 C 语言进行单片机编程的人都遇到过这样的麻烦，想用指针去读写某个地址，如果这个地址是数组没有什么问题，但如果这个地址是用数字 0x1000 这样表示的绝对地址，编译的时候总提示出错，从代码看又好象没有什么错误，搞得是一头雾水，不胜其烦。

```
char *p;
char data_buf[1024];
p=data_buf;                    //这样给指针 p 赋值正确
p=&data_buf[0];                //这样给指针 p 赋值编译出错
```

p=0x1000;　　　　　　　　　　//这样给指针 p 赋值编译出错

要解决这种问题其实很简单，将出错的语句改成下面形式，编译器则不会报告错误。

p=(char*)&data_buf[0];　　　　//编译正确

p=(char*)0x1000;　　　　　　//编译正确

造成上面编译无法通过的原因是赋值操作等号两侧的类型不一致，C 语言语法要求赋值操作时两侧的数据类型必须一致，否则就会告警或报错。C 语言这样处理可以防止程序不同类型数据混用时产生程序员未预料到的逻辑错误，比如上面 p 定义成 char 类型的指针，虽然指针地址也是一个数字，但不能将起等同数字，p=0x1000 这样的语句存在表述不清的问题，因为指针有许多种，光用一个 0x1000 数字我们不能确定这个数字代表的到底是 char 还是 short 或其它类型的指针，加上(char*)后我们就能明确其具体含义。

对于编译器对于赋值操作等号两侧的类型不一致会出错这一结论我们可以多做一些验证，比如有两个不同类型的指针，当把一个指针的地址赋给另外一个指针时同样也会报错。

```
char *p1;
short *p2;
p2=(short*)0x1000;
p1=p2;                    //编译出错
```

改成下面形式：

```
p1=(char*)p2;             //编译正确
```

象(char*)p2/(char*)0x1000 这样的用法就是强制转换，当编译器遇到这样的处理语句后就知道这个地方程序员已经发现数据类型可能不一致，了解数据类型转换可能产生的影响（比如将浮点数转为整型会丢失小数部分），编译器可以进行转换操作后一并编译，如果不加这样的语句编译器会担心程序员没有考虑到数据转换产生的影响而报告错误。

强制转换是一种非常实用的操作，刚接触单片机 C 语言编程的人可能还不会感觉到这类操作的必要性，随着实际开发项目的增多就会逐渐感受到实在在哪些方面，比如常常会发现所写的程序有一大堆警告，但编译依然能通过，这些警告错误相当大一部分都可以用强制转换操作消除掉。

可能有人会有这样的想法，只要编译器编译通过，有几个警告错误无所谓，这种想法不可取，一个好的 C 程序应该是没有任何警告，每一个警告都代表编译器发现一个存在风险的位置，只是编译器也认为这个风险对程序的影响可能比较小，所以编译通过，良好的习惯是当自己的程序编译有警告产生时，应该找出警告的原因，警告往往比错误更难找原因，所以开始可能需要花较多的时间来消除这些警告，要知道警告的原因就那么多，只要积累一定经验后查找警告原因就会变容易。

来看一个浮点数转整型时产生的警告：

```
long x;
float y;
y=(float)3.14;
x=y;                      //将浮点类型的变量赋置给整型变量，产生警告
```

改成下面形式，警告消除。

x=(long)y; //添加(long)强制转换后警告被消除

实际上有无(long)强制转换最终编译产生的代码是一样的，最后 x 都是等于 3，所以编译器这里只是警告，并不报告错误，那警告有什么作用呢？可以提醒程序员留意这里，不要在后面把 x 当作 3.14 继续使用，应该是 3，加了(long)强制转换编译器就理解为程序员发现了 3.14 和 3 的区别，不用再用警告来提醒程序员留意这个地方。

强制转换的另一个好处是可以让程序员对数据类型的控制更加灵活透明，用汇编语言编程的时候各种不同数据类型之间的转换需要自己按照数据类型的格式进行处理，这样虽然程序员在程序中随时都清楚自己需要处理的数据类型，但数据类型之间的转换操作用汇编实现繁琐而复杂，也容易出错，C 语言的强制转换将类型转换的工作交给编译器完成，可以让程序员省心不少。

继续浮点数强制转整型的测试：

```
long x;
float y;
y=(float)3.14;
x=y;                    //执行完这句 x 值为 3，编译器对于这行有警告提示
x=(long)y;              //执行完这句 x 值为 3，无警告提示
x=*((long*)(&y));       //执行完这句 x 值为 0x4048F5C3
```

对于这个例子中的前两行赋值操作不存在什么疑问，都得到我们所预期的整数结果 3，但最后一行的赋值操作 x=*((long*)(&y))却得到 0x4048F5C3 这样一个有趣的结果，为什么会和前两行的赋值操作所得的结果不相同呢？让我们来做一个对比分析。

```
24:    y=(float)3.14;
0040109E    mov    dword ptr [ebp-110h],4048F5C3h
25:    x=y;            将4048F5C3h存入y所在位置[ebp-110h]
004010A8    fld    dword ptr [ebp-110h]从y所在位置[ebp-110h]取输入参数
004010AE    call   __ftol (004011ac) 调用浮点转长整型函数
004010B3    mov    dword ptr [ebp-114h],eax
26:    x=(long)y;    将转换结果存入x所在位置[ebp-114h]
004010B9    fld    dword ptr [ebp-110h]
004010BF    call   __ftol (004011ac)
004010C4    mov    dword ptr [ebp-114h],eax
27:    x=*((long*)(&y));
004010CA    mov    eax,dword ptr [ebp-110h]从y所在位置[ebp-110h]取出长整型内容
004010D0    mov    dword ptr [ebp-114h],eax将取得的内容存入x所在位置[ebp-114h]
```

图 4.12.-1 VC 强制转换结果图

对比编译器产生的汇编代码可以看出 x=y 与 x=(long)y 所产生的码是全一样，都是调用了一个浮点数转整型的函数进行转转，x=*((long*)(&y))则不同，并没有调用浮点数转整型的函数，是直接将 y 在 RAM 中的地址取出来，然后将这个地址作为一个 long*的指针取去里面的内容。

这里我将 x=*((long*)(&y))的操作步骤分解出来：

a. &y 是取存放数据 y 的地址。

b. (long*)(&y)是将取得的地址转换成一个 long*类型的指针。

c. *((long*)(&y))取出这个指针所指的内容。

从这些步骤可以看出整个过程并没有数据强制转换，只是强制转换了一个指针类型，这个转换并不会改变 RAM 中的内容，0x4048F5C3 如果按浮点数格式代表 3.14，如果按长整型则就是十六进制 0x4048F5C3 自己，所以最后一行得到的结果和前两行赋值操作得到的结果不同。

如果我们另外执行 x=*((float*)(&y))的操作，此时 x 又会得到 3 的结果，这里不给出汇编验证的结果，有兴趣的朋友可以自己进行验证。

4.13. 高效实用位运算

标准 C 自身提供有移位、逻辑与或非等位运算指令，但对于一名用 C 语言编写电脑应用程序的程序员，他们往往不太习惯使用 C 语言的位运算功能，因为他们编写程序时候所面对的硬件平台是电脑，电脑所提供的 RAM 资源是单片机无法相提并论的，而且电脑应用程序编写强调的是硬件无关，不需要程序员去了解控制硬件设备，这样使得他们对位运算的依赖性非常低，自然而然就会少用到位运算。

单片机程序则不一样，单片机程序尤其是底层驱动程序需要直接控制硬件，对硬件的控制主要是是设置硬件的相关控制寄存器，这些寄存器往往是一个位就控制一种功能，所以单片机程序可以说是离开了位运算操作真就不行。来看一个 MCU 进行 IO 输入输出功能选择的寄存器，该寄存器控制 IOD 的 8 条 GPIO 口，bit0~bit7 每一个位对应控制一条 GPIO，如果我们想将某条 GPIO 设置成输出，只要将对应的控制位设为 1 即可。

P_IOD_ OutputEn　　　　0x112000C0　　　　IOD Output Enable Register

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Function	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Default	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function	-	-	-	-	-	-	-	-	IOD_Output_Enable							
Default	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Function	Type	Description	Condition
[31:8			Reserved	

]				
[7:0]	IOD_Output_Enable	R/W	IOD Output Enable	0 = Disable 1 = Enable

图 4.13.-1　IO 控制寄存器表

如果不使用位操作指令，就只能是程序知道所有 8 条 GPIO 的输入输出选择，需要更改设置时候将这 8 个控制位一同设为新状态。这样做对于程序员来说无异是一场噩梦，会让程序变得晦涩难懂，一不小心就有可能出错。使用位操作指令可以很好的解决这个问题，比如上面的 IOD，我们现在需要将其中的 IOD3 更改输入输出选择，使用位操作可以让程序一目了然。

定义寄存器指针：

```
#define P_IOD_OutputEn (volatile unsigned long *0x112000C0)
```

定义 bit3 的宏，采用 1 向左移位 3 位的形式更直观：

```
#define BIT3 (1<<3)
```

将 IOD3 设为输出，先读出 IOD 的所有 8 个输入输出控制位，然后通过位或运算将 bit3 设为 1，再将新的设定值回控制寄存器，这样就只将 IOD3 设为输出，其它 GPIO 输入输出状态保持不变：

```
*P_IOD_OutputEn=BIT3|(*P_IOD_OutputEn);
```

将 IOD3 输入和设为输出一样，只是将 bit3 清 0，其它位保持不变：

```
*P_IOD_OutputEn=(~BIT3)&(*P_IOD_OutputEn);
```

另外一些小的 MCU 所能提供的 RAM 资源有限，可能只有几十个字节的 RAM 可供单片机程序员使用，对于程序中只需要 0 和 1 两种状态的状态标志最好是用单个位来表示，否则容易出现 RAM 资源不够用的情况。

标准 C 在位操作方面功能相对有限，比如前面更改 IOD3 的输入输出控制，需要经过“将原设定值从寄存器读出→与/或操作需要设定的位得到新设定值→将新设定值写回寄存器”这三步才能完成，显然代码效率并不高，如果能够直接一步就完成 IOD3 的设定，无疑是一个不错的选择，标准 C 在这方面就存在不足，虽然可以通过位结构来定义位，但实质上只是从语法上看上去简单一些，在汇编层面看还是需要前面三步通过与非操作来设定相应位。

下面定义了一个位结构。

```
struct{
  unsigned incon: 8;          //incon 占用低字节的 0~7 共 8 位
  unsigned txcolor: 4;        //txcolor 占用高字节的 0~3 位共 4 位
  unsigned bgcolor: 3;        //bgcolor 占用高字节的 4~6 位共 3 位
```

```
    unsigned blink: 1;          //blink 占用高字节的第 7 位
}ch;
```

对于 ch.blink=1 这样的操作，编译器得到的汇编指令等同 ch=BIT7|ch，因为电脑的 CPU 和现在一些使用 ARM 内核的 MCU 都没有提供可以将单个位置 1 或清 0 的指令，为了提供良好的通用性，编译器处理这类代码会得到同样的汇编指令。

那是不是单片机的 C 语言程序只能按这种三步走的方式来设置控制位了呢？答案是否定的，为了得到更好的代码效率，不少小的 MCU 都支持位操作，也就是有一些 MCU 支持将单个位置 1 或清 0 的汇编指令，对于这类 MCU，如果支持 C 编程那么厂商提供的 C 编译器肯定也支持单个位置 1 或清 0 的操作，编译器在标准 C 的指令之外提供了额外的指令来支持位操作。

比如最常见的 51 系列的 MCU 就支持单个位置 1 或清 0，来看看用 KEIL C 的编译器是怎么样对单个位进行操作的。

KEIL C 在标准 C 之外提供 sbit 指令，用来定义位变量。

```
sbit MCU_LED0 = P1^2;          //将 GPIO P1.2 定义成为 MCU_LED0，P1 寄存器为 90H
MCU_LED0=1;                    //P1.2 输出 1，对应汇编指令 SETB 90H.2
MCU_LED0=0;                    //P1.2 输出 0，对应汇编指令 CLR 90H.2
```

注意这里是将 MCU_LED0 用 sbit 直接定义成位变量，和位结构中的位变量完全不同，当编译器遇到 MCU_LED0 就会将当前的 C 代码用 51 的位操作汇编指令实现。

不是所有的单片机都直接支持直接对单个位置 1 或清 0 的操作，对于一些采用通用编译器和 CPU 内核的单片机为了平台的一致性，大都不支持这种做法，比如使用 ARM 内核的各种 MCU 就不支持直接对单个位置 1 或清 0 的操作，这类 MCU 还是必须用三步走的方式来完成。

这样看来对单个位置 1 或清 0 要想做到高效还得需要 MCU 自身有相应位操作指令支持，并不是对所有 MCU 都有效，那有哪些方面可以体现位操作的高效性呢？让我们来看看在乘除运算方面的影响。

有的 MCU 自己支持乘法除法指令，但乘法和除法指令相较其它指令往往会占用更多的指令周期，如果 MCU 不支持乘法除法指令，那就需要用软件方法来实现，更为缓慢，这样对于 C 语言中的乘除运算 MCU 会占用较多的时间执行，总体上说对于乘除运算处理 MCU 难以高效。

有些特殊的乘除运算我们可以用位操作的移位指令来实现，这样做可以将乘除运算转为高效的位运算，比如 x=x/2 和 x=x*8，如果用 x=x>>1 和 x=x<<3 来实现的话效率肯定会更高。

除了可以让程序变得更加高效外，使用位操作指令还可以让程序简洁可靠，如果不用移位指令，要对某个数据位进行定义就必须先将该位为 1 其它位全为 0 的十六进制数人工计算出来，比如现在要定义 bit11，就得知道 bit11 为 1 其它位为 0 的十六进制数是 0x0800，然后#define BIT11 (0x0800)。如果只对单个位进行定义对于二进制和十六进制转换关系熟悉的程序员还不是难事，但如果现在需要定义 32bits 中的 2、9、15、21、28 这些位的组合呢？恐怕就有点麻烦，稍有不慎就可能定义出错。

来看一下利用了移位操作指令的情况：

```
#define BIT7 (1<<7)
#define BIT_2_9_15_21_28 ((1<<2)|(1<<9)|(1<<15)|(1<<21)|(1<<28))
```

是不是变得非常简单，复杂的二进制和十六进制转换工作由编译器完成，想定义错误都难啦。也许会有朋友担心这样做会不会产生代码效率变低的问题，从代码看这样的宏定义包含不少移位操作，按照编译原理编译时会用这个定义替换后面代码中相应的宏，那只要是用到这样宏的地方都会包含这些移位操作，和前面直接定义成十六进制数相比好像代码效率要低。有这种顾虑说明对 C 语言的宏有了一定了解，在实际应用中不用担心这个问题，现在稍微做得好一点的编译器都考虑到了这些问题，编译器会尽量让编译出来的汇编指令简洁高效，象上面的宏 BIT_2_9_15_21_28 编译器会先判断所定义的内容是否可以优化，如果能优化优化出简洁的表达式，这样对于位运算表达式((1<<2)|(1<<9)|(1<<15)|(1<<21)|(1<<28))会被预先处理为 0x10208204，然后用 0x10208204 替换后面宏 BIT_2_9_15_21_28。

单片机程序员常会遇到这样一道关于位操作的笔试题，请写一个函数将数实现高低位交换，比如现在有一个 16bits 的整型数，该函数将其 bit0 与 bit15 交换、bit1 与 bit14 交换......，题目并不难，但却能难倒不少对 C 语言语法还算熟悉的人，究其原因就是不能熟练使用 C 语言有关位操作运算的指令。

这里给出两种利用位操作运算实现的函数代码，以便加深对位操作运算作用的理解。

函数一：

将数据从左向右、结果从右向左进行移位，每次移位先判断数据的最高位，如果该位为 1 则将结果的最高位也置 1，否则清 0，循环 16 次之后实现要求。

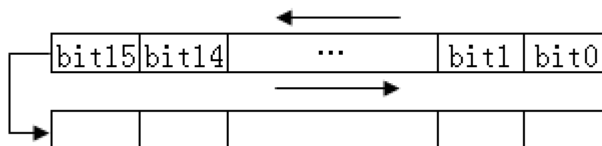

图 4.13.-2　移位操作示意图一

```
#define BIT15 (0x8000)          //定义最高位 bit15 的宏
void swap_bits(unsigned short *data)
{
  unsigned short data_temp,i;
  data_temp=0;                  //先将结果所有位清 0
  for(i=0;i<16;i++)
  {
```

```
    data_temp>>=1;            //结果右移一位
    if((*data)&BIT15)         //如果数据最高位为 1 则结果最高位置 1
    {
      data_temp|=BIT15;
    }
    *data<<=1;                //数据坐移一位
  }
  *data=data_temp;            //返回结果
}
```

函数二：

从最低位开始依次判断数据的每个位，如果该位为 1 则将结果中与之相映射的位置 1，否则清 0，循环 16 次后实现要求。

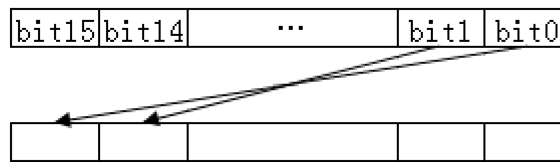

图 4.13.-3　**移位操作示意图二**

```
void swap_bits(unsigned short *data)
{
  unsigned short data_temp,i;
  data_temp=0;                //先将结果所有位清 0
  for(i=0;i<16;i++)
  {
    if((*data)&(1<<i))        //判断当前的第 i 位是否为 1
    {
      data_temp|=1<<(15-i);   //为 1 则将结果中与之相映射的位置 1
    }
  }
  *data=data_temp;            //返回结果
}
```

4.14. 宏和 register

只要接触过 C 语言，大都熟悉宏的作用，即便是汇编语言，同样也有宏的概念，所以宏对于单片机程序员来说并不陌生。虽然大多数人不陌生宏，但能将宏用好的人并不多见，宏的优点不少书籍都有详细说明，这里我不再累述，而是通过一些例子来让大家感性的学习宏的使用。

1.宏不要使用小写字母

这一点是定义宏时的第一基本准则，目的是为了在程序中能将宏和函数、变量很明显进行区分，使用小写字母虽然不会产生错误，但对程序的易读性有负面作用，这是一条约定俗成的规则，没有太多的理由，如果你想用好宏，就必须将这条谨记在心。

2.程序中的各种定义尽量用宏

不少人都遇到过这样的事情，程序写得已经差不多了，发现某些地方存在问题需要修改，可能只需要修改一两种操作，可这一两种操作分布在程序的不同位置，于是就要一个一个的找出来，这个工作比写程序还麻烦，一不小心就会出现漏改的情况，许多时候造成这种麻烦的原因都是程序员没有很好的使用宏。

来看一个例子，一个简单的 MCU 提供 pa0~pa7 共 8 条 IO 口，最初将 pa2 定为输入按键，pa5 为输出 LED 控制信号，程序已经写好后突然提出新要求，要求硬件和另外一款芯片兼容，这样就需要交换 pa2 和 pa5 的功能才能做到。

如果不使用宏，这个修改就需要将程序里面所有关于 pa2 和 pa5 的操作找出来，然后逐个交换，这样需要修改的地方可能有几十甚至上百。如果使用了宏呢？情况则大不一样。

只需要将

```
#define LED_OUTPUT_ENABLE()    (_pa5=1)
#define LED_OUTPUT_DISABLE()   (_pa5=0)
#define IS_KEY_RELEASED()      (_pa2)
```

改为

```
#define LED_OUTPUT_ENABLE()    (_pa2=1)
#define LED_OUTPUT_DISABLE()   (_pa2=0)
#define IS_KEY_RELEASED()      (_pa5)
```

从原来的几十甚至上百个地方需要修改一下就变成了只需要修改三处，工作量大为减少，而且因为程序中相关操作都是使用宏来表述，完全不用担心改错。

既然对系统硬件资源的定义最好使用宏，代码中常用做比较等操作常量同样也最好是使用宏，再来看一个常量使用宏的例子。

```
#define VOLTAGE_12V6   (96)
#define VOLTAGE_12V3   (94)
#define VOLTAGE_12V0   (91)
#define CURRENT_300MA  (96)
#define CURRENT_250MA  (80)
#define CURRENT_200MA  (64)
```

同样为 96 的值，有的地方可能是用于电流比较，有的地方却又是用做电压比较，如果不用宏的话两者就会混在一起，当程序员需要修改电流比较用的 96 的时候，即使用了编辑工具的查找功能，还需要人为检查是不是电压比较，否则就会出错。用了宏则和前面硬件定义一样，只要在定义宏的地方修改一下就行，基本上不用再去管程序中的宏。

当然使用宏也不代表完美，当程序量比较大而且功能复杂时，需要写出大量的宏才能满足需求，这样就会让程序编写要麻烦一些；另外还有一些功能有时候用宏来实现也不是很方便，有时候遇到问题后要用不同方式尝试，这时会在程序中先用一些临时测试代码进行尝试，如果要让临时测试代码的增减与相关宏同步会让程序员感觉更繁琐；再就是调试的时候宏对应的代码往往不能单步跟踪，一执行就整个宏。

相较优点宏的不足显然只占极少部分，所以写程序的时候一定要养成多使用宏的习惯，当你代码写得足够多的时候就会越发感受到宏给你带来的便利。

3.防止头文件被重复包含

```
#ifndef MYHEADER_H
#define MYHEADER_H
  //头文件内容
#endif
```

示例：

头文件 myheader.h 内容

```
//#ifndef MYHEADER_H
//#define MYHEADER_H

//#endif
```

头文件 mydrv.h 内容

```
//#ifndef MYDRV_H
//#define MYDRV_H
void mydrv1(void);
void mydrv2(void);
//#endif
```

头文件 myapp.h 内容

```
//#ifndef MYAPP_H
//#define MYAPP_H
void myapp(void);
//#endif
```

文件 mydrv.c 内容

```
#include “mydrv.h”
void mydrv1(void)
{
  //代码
}
void mydrv2(void)
{
  //代码
}
```

文件 myapp.c 内容

```
#include “mydrv.h”      //需要引用该头文件申明 mydrv()以便在程序中调用
#include “myapp.h”
void myapp(void)
{
  //其它代码
  mydrv2();
  //其它代码
}
```

文件main.c内容

```
#include “mydrv.h”      //需要引用该头文件申明mydrv1()以便在程序中调用
#include “myapp.h”      //需要引用该头文件申明myapp()以便在程序中调用
void main(void)
{
  //其它代码
  mydrv1();
  myapp();
  //其它代码
}
```

编译器编译上面main.c的时候就会产生重复申明的错误，在主程序main.c的头两行分别引用了头文件mydrv.h和myapp.h,留意在myapp.h一开始也有引用头文件mydrv.h,这样就在将mydrv.h重复引用了两次，形成将mydrv1()和mydrv2()申明两次的错误。

如果将上面代码中的注释去掉，重复申明的错误将被消除，编译器在main.c的第一行先将头文件mydrv.h引用进来，此时发现并没有定义MYDRV_H，于是进行定义，当处理第二行的myapp.h再次需要引用mydrv.h时，不同的编译结果产生，因为此时MYDRV_H已经被定义，所以mydrv.h中间的内容全部被跳过不进行再次编译，从而也就不会产生重复申明的错误。

4.用宏定义表达式时，要使用完备的括号防止出错

```
#define ADD(a,b)  (a+b)
#define ADD(a,b)  a+b
```

有括号ADD(2,3)*4=(2+3)*4=20

无括号ADD(2,3)*4=2+3*4=14

显然无括号情况得到的结果与我们所期望的不一致。

好像(a+b)的方式再不存在什么问题，其实不然，还有潜在风险隐藏在里面，对于ADD(a,b)的例子可能不会出错，另外一个例子则不一样。

```
#define SQUARE(x)  (x*x)
```

SQUARE(1+2)=1+2*1+2=5

显然我们是想得到1+2结果的平方，正确结果应该是9，现在错误的得到5。

```
#define SQUARE(x)  ((x)*(x))
```

SQUARE(1+2)=(1+2)*(1+2)=9

这次得到的结果是正确的，所以为了防止宏定义出错，必须将表达式中的参数全部用括号括起

来。

正确的宏定义方式：

```
#define ADD(a,b)  ((a)+(b))
#define SQUARE(x)  ((x)*(x))
```

5.使用宏时不允许参数发生变化

接着看用来求平方的宏 SQUARE。

```
#define SQUARE(x) ((x)*(x))

int a=2;
int b;
b=SQUARE(a++);    //SQUARE(a++)=(a++)*(a++)，结果执行了两次加一操作得到 a=4
```

正确的用法是：

```
b=SQUARE(a);
a++;              //只执行一次加一操作得到 a=3
```

6.宏所定义的多条表达式应放在大括号中

下面的语句只有宏的第一条表达式被执行：

```
#define INTI_VALUE(x,y)\
  x=1;\
  y=2;

  for(i=0;i<TOTAL_NUM;i++)
    INTI_VALUE(buf[i][0], buf[i][1]);
```

这里的 for(;;)循环语句展开后为

```
  for(i=0;i<TOTAL_NUM;i++)
  buf[i][0]=1;\                //此处反斜杠被编译器理解为接下一行
  buf[i][1]=2;
```

for(;;)循环因为 buf[i][0]=1;后面的分号而将 buf[i][1]=2 排除在循环体内。

正确的用法应为：

```
#define INTI_VALUE(x,y)\
{\
  x=0;\
  y=0;\
}

for(i=0;i<TOTAL_NUM;i++)
{
  INTI_VALUE(buf[i][0], buf[i][1]);
}
```

注：这里的反斜杠\表示下一行继续为宏定义的内容

7.将自己常用的类型重新定义，防止由于平台不同而产生的类型字节数差异，方便移植

```
typedef unsigned long   UINT32;
typedef unsigned short  UINT16;
typedef unsigned char   UINT8;
typedef signed char     INT8;
typedef signed long     INT32;
typedef signed short    INT16;
```

在程序中用后面定义的新类型名来定义变量类型，比如现在需要定义一个 8bits 的单字节无符号数，应定义为 UINT8 x。这种定义的优点直接说明了变量的数据位宽，程序员在编写程序时不容易出现数据位宽混淆的错误。如果需要将程序移植到另外的 MCU 平台上，即使编译器对数据类型的定义不一致也很容易移植，只要将这些宏定义更改成正确的类型即可。

8.使用宏进行跟踪调试

程序员在对程序调试时有时候并不想通过调试器设置断点，因为程序一旦执行到断点位置，虽然可以用调试器去查看 MCU 的工作状态，但这么做不能让程序处于连续不间断工作状态，所以断点调试和真实的程序运行状态还是存在一定差异，用宏输出调试信息就可以让程序更加接近实际工作状态。

```
#define DEBUGMSG(msg)  \
{\
#ifdef _DEBUG_\
```

```
printf(msg);\
#endif\
}
```

这样在程序中只要定义是否打开 DEBUG 调试功能的宏_DEBUG_就可以在程序中选择是否打开调试输出功能，程序调试完毕需要发放程序时关闭宏_DEBUG_所有的 DEBUGMSG()都被编译器处理成空操作。

9.宏定义里面的#和##

通常我们使用#把宏参数变为一个字符串，用##把两个宏参数贴合在一起。

```
#define STR(s)  #s
#define CONS(x,y)  int(x##f##y)

printf(STR(123));        //输出字符串"123"
printf("%d",CONS(2,6));  //2f6 输出 758
```

这类宏我比较少用，个人也建议少用，因为理解起来相对要晦涩一些。

10.一些实用的宏示例

求最大值和最小值

```
#define MAX(x,y)  (((x)>(y))?(x):(y))
#define MIN(x,y)  (((x)<(y))?(x):(y))
```

高低字节操作（可以不用&0xFF，类型为自定义类型）

```
#define HI_BYTE(n)  (UINT8)((n>>8)&0xFF)
#define LO_BYTE(n)  (UINT8)(n&0xFF)
#define BYTE2WORD(hi,lo)  (UINT16)((hi<<8)|lo)
```

高低字操作（可以不用&0xFFFF，类型为自定义类型）

```
#define HI_WORD(n)  (UINT16)((n>>16)&0x0FFFF)
#define LO_WORD(n)  (UINT16)(n&0x0FFFF)
#define WORD2DWORD(hi,lo)  (UINT32)(((UINT32)hi<<16)|lo)
```

读写指定地址上的字节

```
#define MEM_BYTE(x)  (*((UINT8 *)(x)))
```

字母大小写转换

```
#define UPCASE(c)  (((c)>='a'&&(c)<='z')?((c)-0x20):(c))
#define LOWCASE(c)  (((c)>='A'&&(c)<='A')?((c)+0x20):(c))
```

十进制和 BCD 转换

```
#define BCD_TO_DEC(bcd)  ((((UINT8)(bcd))>>4)*10+(((UINT8)(bcd))&0x0f)))
#define DEC_TO_BCD(dec)  (((((UINT8)(dec))/10)<<4)|((UINT8)(dec))%10))
```

返回数组元素的个数

```
#define ARR_SIZE(a)  (sizeof((a))/sizeof((a[0])))
```

位操作

```
#define TEST_BIT(x,offset)  (1&((x)>>(offset)))
#define SET_BIT(x,offset)  ((x)|= (1<<(offset)))
#define CLR_BIT(x,offset)  ((x)&=(~(1<<(offset))))
```

面试常问的用宏表示一年有多少时间（留意溢出）

```
#define MINS_OF_YEAR  ((UINT32)(365*24*60))
#define SECS_OF_YEAR  ((UINT32)(365*24*60*60))
```

一组有关 8/16/32bits 数处理的宏

```
typedef union
{
    UINT16      WordCode;
    struct
    {
        UINT16  _byte0      : 8;    //LSB byte code
        UInt16  _byte1      : 8;    //MSB byte code
    } ByteCode;
}UWORD;
typedef union
{
    INT16       WordCode;
    struct
    {
```

```
        UINT16  _byte0      : 8;    //LSB byte code
        INT16   _byte1      : 8;    //MSB byte code
    }ByteCode;
}SWORD;
typedef union
{
    UINT32      LongCode;
    struct
    {
        UINT16  _word0      :16;    //LSB word code
        UINT16  _word1      :16;    //MSB word code
    }WordCode;
    struct
    {
        UINT16  _byte0      :8;      //bit 7  ~  0
        UINT16  _byte1      :8;      //bit 15 ~  8
        UINT16  _byte2      :8;      //bit 23 ~ 16
        UINT16  _byte3      :8;      //bit 31 ~ 24
    }ByteCode;
} ULONG;
typedef union
{
    INT32       LongCode;
    struct
    {
        UINT16  _word0      :16;    //LSB word code
        INT16   _word1      :16;    //MSB word code
    }WordCode;
    struct
    {
        UINT16  _byte0      :8;      //bit  7 ~  0
        UINT16  _byte1      :8;      //bit 15 ~  8
        UINT16  _byte2      :8;      //bit 23 ~ 16
        INT16   _byte3      :8;      //bit 31 ~ 24
    }ByteCode;
```

```
}SLONG;
#define BYTE0          (ByteCode._byte0)
#define BYTE1          (ByteCode._byte1)
#define BYTE2          (ByteCode._byte2)
#define BYTE3          (ByteCode._byte3)
#define WORD0          (WordCode._word0)
#define WORD1          (WordCode._word1)
#define ByteCode(x)    ((x._byte1<<8)|x._byte0)
#define WordCode(x)    ((x._byte3<<24)|(x._byte2<<16)|(x._byte1<<8)|x._byte0)
#define Byte0(x)       (x._byte0)
#define Byte1(x)       (x._byte1)
#define Byte2(x)       (x._byte2)
#define Byte3(x)       (x._byte3)
```

讲完宏再讲另外一个很有用的伪指令 register，使用 register 在提高代码执行效率方面有着强大的威力，但要注意的是一些简单 MCU 的编译器可能不支持该指令。

为什么 register 指令在提高代码执行效率方面存在着优势呢？原因需要从 regsiter 指令本身含义开始找，该指令表示变量不是在 RAM 中申请，而是使用 MCU 的通用寄存器，这样程序在处理用 register 定义的变量的时候就不需要在访问 RAM，直接读写寄存器就能完成任务，可以将代码效率提高。

使用 register 指令存在着限制条件，首先因为这种变量占用了 MCU 的通用寄存器，如果一直占用这些寄存器的话，其它代码会没有可用的通用寄存器而无法执行，所以 register 变量只能是动态局部变量和函数参数可以使用，以避免某个变量长期占用通用寄存器；其次任何 MCU 的通用寄存器个数都不多，数目有限，所以一段代码同时能支持的 register 变量也相应有限，如果申请的个数太多，编译器会将超出的 register 变量处理为普通 RAM 变量。

4.15. 手机里的计算器

在没有接触到数学的极限概念之前，不少人都会为这样的问题困扰：1/3=0.33333333…，1/3*3=1，可是另外却有 0.33333333…*3=0.99999999…，0.99999999…明明不是 1，但通过这三个等式却得出 0.99999999…等于 1 的结论，让人着实有些困惑，数学里的极限概念为我们解释了这个困惑。

这个问题好像和单片机 C 语言编程无关，实质上两者见也确实没有任何直接关系，在这里提出来只是为了引申出一个单片机 C 语言编程的问题：数据类型的处理。C 语言常用的数据类型在我看

来可以分成两类，浮点和整型，浮点就是小数，整型顾名思义就是整数。就单片机来说，许多时候都用不上浮点数，只需要整数即可满足应用要求，年长日久，使得许多单片机程序员对浮点数陌生起来，在一些特定的应用需要使用小数时，程序编写就可能不够理想。

让我们来看一个需要使用到小数的应用实例，无论是电脑里面的计算器还是街上可以买到的计算器，当你输入 1/3 之后可以得到 0.333333…的结果，如果此时你接着输入*3，显示的结果会明确无误的告诉你为 1，看来微软的工程师和专业做计算器芯片的工程师在数制处理上不存在什么问题。

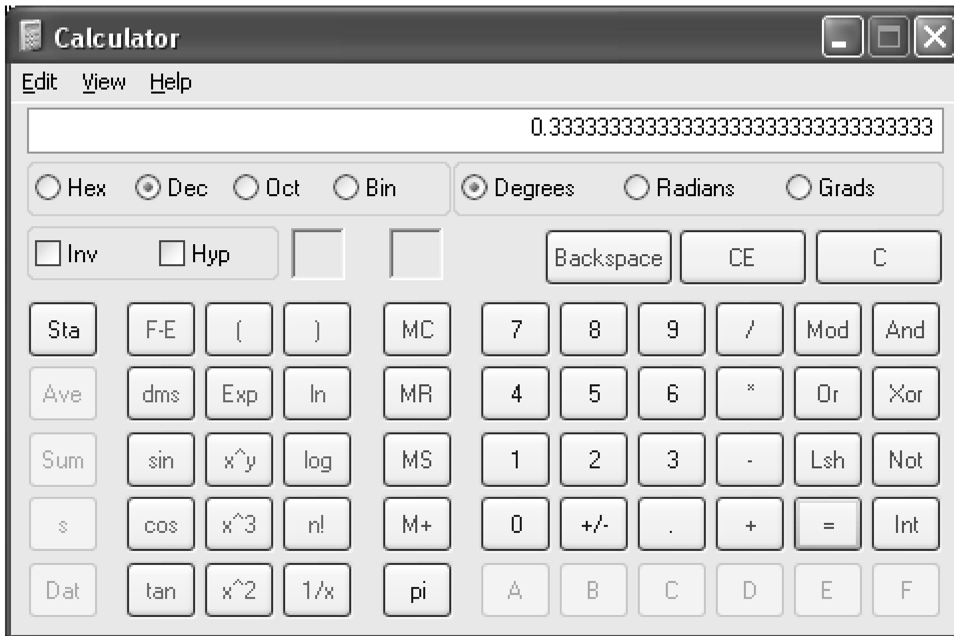

图 4.15.-1　电脑计算器图

手机实质是一个用单片机实现以通讯功能为主、内部带有其它若干附加功能的电子产品，它里面的程序自然也就是单片机程序。现在的手机基本上都带有简单的计算器功能，让我们来看看手机里面的计算器功能怎么样？结果可能会让你失望，当输入 1/3*3 之后，会看到显示的结果并不为 1，而是 0.99999999，可能这个结果在实际生活中影响不是很大，毕竟手机的计算器功能只是帮助使用者算个帐什么的，不需要太高精度，但从数学的角度看这无疑是一个错误的结果。

不要认为只是山寨的手机才有上面的问题，象在知名的 N 记、M 记手机中这个问题同样存在，我没有做过手机软件的开发，或许在设计、编写手机程序的工程技术人员眼里这也许不是一个问题，但在我看来这起码是一点不足，一个程序员应该尽可能的让自己实现的功能与已有的标准一致，除非是自己的创新。

因为自己不在手机行业，所以只能推测造成这个现象的原因，既然电脑和真正的计算器都可以做到正确显示 1，我想单片机程序同样也能够实现。不过这只是我个人的观点，如果想要别人认可我的观点，就需要更多的证据，可没有手机可以让我们自己编程来进行验证，没关系，有变通的方法，只要用 C 语言在电脑上验证算法就行。

我个人猜测手机中计算器计算结果不够精确是没有采用浮点数做为中间变量，为便于对验证过程做出解释说明，验证程序并不写成真正的计算器功能，只是对除法结果为无穷小数的情况选出几种情况将计算过程进行验证，因此在验证程序中会用浮点数和整型数做对比，计算过程显示8位有效数字。

程序编译环境：VC6.0 Console App32 模式

验证思路：因为32bits浮点数只能提供6位有效数字，验证过程需要提供8位有效数字，所以用一个整型变量存放去处小数点后的整数运算结果，另外一个整型变量存放小数点位置，另外还有一个浮点变量，直接浮点运算的结果存放在这个变量中。整数模式只利用两个整型变量，显示时通过这两个变量将运算结果的实际8位有效数字显示出来；浮点模式显示的时候还需要依据计算结果做一些特殊判别以确定使用整数结果还是浮点结果显示，这里没有将这部分程序加进来。

```
#include "stdafx.h"
int exp(int x,int y)//未调用VC库函数，该函数实现x的y次幂运算
{
    int ret;
    ret=1;
    while(y--)
    {
        ret=ret*x;
    }
    return ret;
}
int main(int argc, char* argv[])                        //验证a/b*b过程
{
    char string[256];                                   //用来显示计算过程中的数字
    signed long result_int;                             //整数模式保存计算过程结果
    signed long radix_point;                            //整数模式保存小数点位置
    float result_float;                                 //浮点模式保存计算过程结果
    int x,y,a,b;
    int i;
    a=1;                                                //验证用被除数
    b=3;                                                //验证用除数
    x=a;
    y=b;
    printf("Int mode calculate (%d/%d)*%d:\n",a,b,b);   //整数模式运算结果
```

```
printf("Step 1. Calculate %d/%d\n",a,b);            //整数模式计算 a/b 过程
result_int=0;
radix_point=0;
i=0;
while(x>=(y*(exp(10,i))))                           //计算整数结果和小数点位置
{
    i++;
}
if(i!=0)
{
    result_int=x/y;
    x=x%y;
}
for(;i<8;i++)
{
    radix_point++;
    result_int=(result_int*10)+(10*x/y);
    x=(10*x)%y;
}
for(i=0;i<256;i++)                                  //清空显示 buffer
{
    string[i]=0;
}
sprintf(string,"%d",result_int);                    //整数结果填入显示 buffer
for(i=0;i<radix_point;i++)                          //整数结果中插入小数点
{
    string[8-i]=string[7-i];
}
string[8-radix_point]='.';
printf("        %d/%d=%s\n",a,b,string);            //显示计算结果
printf("Step 2. Calculate *%d\n",b);                //整数模式计算*b 过程
i=0;
while((result_int*b)>=exp(10,i++));                 //计算整数结果和小数点位置
radix_point=radix_point+8+1-i;
result_int=result_int*b;
```

```
for(i=0;i<256;i++)
{
    string[i]=0;
}
sprintf(string,"%d",result_int);                    //整数结果填入显示 buffer
for(i=0;i<radix_point;i++)                          //整数结果中插入小数点
{
    string[8-i+1]=string[8-i];
}
string[8-radix_point]='.';
for(i=9;i<256;i++)                                  //清除显示 buffer 多余的位数
{
    string[i]=0;
}
printf("        %d/%d*%d=%s\n",a,b,b,string);       //显示整数计算结果
printf("\n");

x=a;
y=b;
printf("Float mode calculate (%d/%d)*%d:\n",a,b,b);//浮点模式运算结果
result_float=(float)x/(float)y;                     //计算浮点结果
result_float=result_float*(float)y;                 //计算浮点结果
printf("Step 1. Calculate %d/%d\n",a,b);            //浮点模式计算 a/b 过程
result_int=0;
radix_point=0;
i=0;
while(x>=(y*(exp(10,i))))                           //计算整数结果和小数点位置
{
    i++;
}
if(i!=0)
{
    result_int=x/y;
    x=x%y;
}
```

```
for(;i<8;i++)
{
    radix_point++;
    result_int=(result_int*10)+(10*x/y);
    x=(10*x)%y;
}
for(i=0;i<256;i++)
{
    string[i]=0;
}
sprintf(string,"%d",result_int);
for(i=0;i<radix_point;i++)
{
    string[8-i]=string[7-i];
}
string[8-radix_point]='.';
printf("        %d/%d=%s\n",a,b,string);            //显示整数计算结果
printf("Step 2. Calculate *%d\n",b);                //浮点模式计算*b 过程
i=0;
while((result_int*b)>=exp(10,i++));                 //计算整数结果和小数点位置
radix_point=radix_point+8+1-i;
result_int=result_int*b;
for(i=0;i<256;i++)
{
    string[i]=0;
}
sprintf(string,"%d",result_int);                    //整数结果填入显示 buffer
for(i=0;i<radix_point;i++)                          //整数结果中插入小数点
{
    string[8-i+1]=string[8-i];
}
string[8-radix_point]='.';
for(i=9;i<256;i++)                                  //清除显示 buffer 多余的位数
{
    string[i]=0;
```

```
    }
    printf("        %d/%d*%d=%d\n",a,b,b,(int)result_float);
                                                            //显示浮点计算结果
    printf("\n");

    return 0;
}
```

运行该程序所得到的 1/3*3 运算结果：

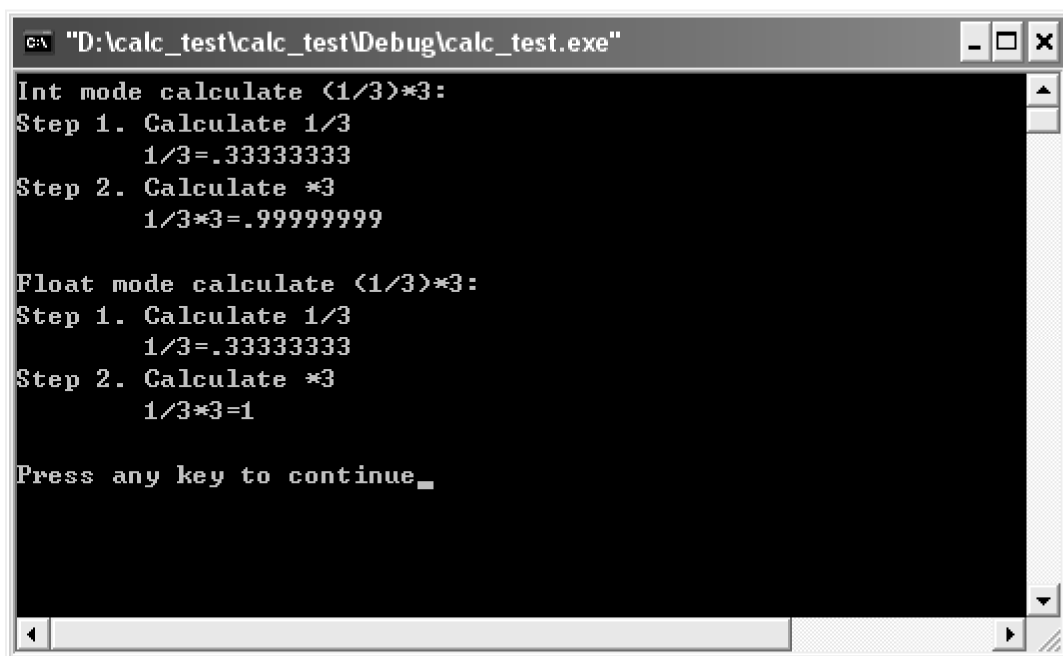

图 4.15.-2　整数模式电脑模拟结果图

运行该程序所得到的 59/23*23 运算结果：

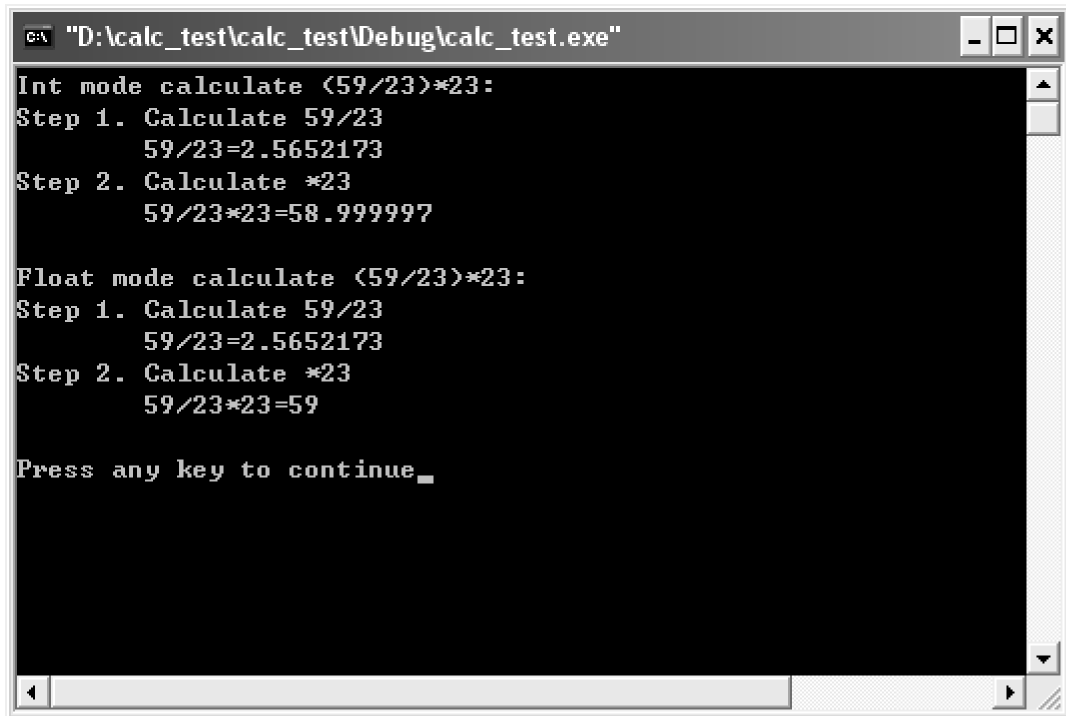

图 4.15.-3　浮点模式电脑模拟结果图

由上面的验证结果看确实可以做到与电脑中的计算器一样的效果，只是实现的程序要复杂一些，对于知名品牌，程序复杂不是一个可以认可的理由，而且手机提供计算器功能已经存在许多年，完全有时间将计算器功能做得更完善，从技术角度看这样的做法不可取。

4.16. 函数设计

想要设计出一个好的函数并没有固定的规律，主要是靠程序员自己的经验积累，基本规则是设计的函数高效、安全、完善、易懂，只要写出的函数做到这几点，就算得上是好函数。

函数一般依据功能分类整理后放在独立的程序文件中，为了让别人一眼就能知道这个文件中间的内容，需要在文件头放置文件注释信息，以方便他人阅读。

```
//------------------------------------------------------------
//Copyright (C), 2000-2009, XXX Co., Ltd.
//Filename:      DmaDrv.c
//Description:  简要说明该程序文件完成什么功能
//Function List:
//               具体函数列表略
//Remark:        注意事项
```

```
//History:
//        1.Date:     2009/07/02
//          Author:   XXX
//          Version: 0.1
//          Modification:
//            First release version.
//        2.Date:     2009/07/24
//          Author:   XXX
//          Version: 0.2
//          Modification:
//            2.1 Remove some member of struct DmaRequest
//            2.2 Add Dma_setCEMode(),Dma_getCEMode()
//------------------------------------------------------------
```

这是我常用的注释方式，里面包含有功能描述、注意事项、作者、版本和日期等信息，这部分注释信息如果想表达得更加清晰准确可以使用中文，有的人可能习惯使用/******** ********/这样的 C 语言注释方式，我习惯在每一条需要注释的行前加双斜线//，与/******** ********/会麻烦一些，但可以避免产生/**** /**** ****/ ****/阴影部分未被屏蔽的错误。

接下来是函数的主体部分，也就是函数的具体代码，我们通过两个函数设计的实例来介绍如何进行函数设计，例子还是选用面试常问的一个题，不调用 C 的库函数实现函数 char * strcpy(char * strDest,const char * strSrc)，另外我们再实现 void* memcpy(void* dest, const void* src, int count)。

先给出面试题的“标准”答案。

```
char* strcpy(char* strDest,const char* strSrc)
{
  if((strDest==NULL)||(strSrc==NULL))
  {
    return NULL;                          //如果源地址或目的地址为空（0）返回空
  }
  char *strDestCopy=strDest;              //保留源地址指针
  while((*strDest++=*strSrc++)!='\0');   //复制直到内容为 0x00，这个 0x00 也要复制
  return strDestCopy;                     //返回源地址指针
}
```

这是标准 C 自己的实现方法，如果 PC 编程从 C 语言角度系统的看这确实是一段优秀的代码，简洁高效，但把这个函数独立出来做为面试题将前面代码当做标准答案我个人看不大妥当，尤其是追问为什么需要返回 char*这个问题后显得更为不妥。

在定制C语言的一系列标准的时候，设计者并没有预计到今天C语言遍地开花的局面，只是基于电脑程序的准则来进行相关设计，函数中源地址或目的地址为空时返回空就是电脑的程序是不允许对地址0进行读写操作的，那个位置属于程序保留区域，一旦修改就会导致程序崩溃。需要返回char*的答案是方便进行链式调用，比如strlen(strcpy(buf1,buf2))，实质上就是当时设计的所有关于字符串的函数都遵循着设计者自己定义返回指的规则，如果是空指针代表出错，非空则表示操作成功，返回的指针为目的指针。

C 语言那时定义的函数是不是就完美了呢，答案是否定的。比如返回参数只能告诉程序员当前调用有没有产生错误，并不能告诉程序员具体的错误类型，加上现在C语言的使用范围早已经远超电脑程序的领域，在单片机应用方面，程序运行不一定继续遵循地址0不能读写的准则，有些特殊应用甚至还特意需要向这个地址进行读写，比如前面章节中提到的程序重载的功能就需要这样操作。

如果你在面试中遇到这个问题而考官继续坚持返回 NULL 和方便链式调用为标准答案，恐怕你要在考官以后在技术方面会不会成为你的良师益友方面多加斟酌。

转回正题，看看这两个函数到底怎么写为好，因为本书是以介绍单片机相关经验为主线，所以接下来这两个函数也会以适合单片机应用的方向进行设计。

假定MCU为32bits，函数原型需要做出修改以满足我返回参数的设计。

```
UINT32 strcpy(UINT8* strDest,const UINT8* strSrc)
UINT32 memcpy(UINT8* dest,const UINT8* src,UINT32 count)
```

函数同样需要有函数说明。

```
//------------------------------------------------------------
//Name:          UINT32 strcpy(UINT8* desBuf,const UINT8* srcBuf)
//Description:   从源地址strSrc复制数据到目的地址strDest
//               遇到内容为0x00结束复制，0x00自己需要被复制
//Input:         desBuf -- 目的地址（从以此地址为起始位置的空间读数据）
//               srcBuf -- 源地址
//Output:        desBuf -- 目的地址（数据写入以此地址为起始位置的空间）
//               srcBuf -- 源地址
//Return:        >0            成功复制的数据字节数
//               ERR_PARAMETER 输入参数错误
//               ERR_EMPTY_STR 复制对象为空的字符串
//Remark:        注意本函数没有溢出保护，使用时候应避免产生溢出
//History:
//        1.Date:    2009/11/12
//          Author:  XXX
```

```
//          Version: 0.1
//          Modification:
//            First release version.
//-------------------------------------------------------------
UINT32 strcpy(UINT8* desBuf,const UINT8* srcBuf)
{
  UINT32 count               //用来累计复制的数据个数
  count=0;                   //将其清0
  if((UINT32)desBuf==(UINT32)srcBuf)
  {
    return ERR_PARAMETER;    //目的地址和源地址相同认为参数错误
  }
  if(*strSrc==0x00)
  {
    return ERR_EMPTY_STR;    //目的地址首字节为0x00认为是空字符串
  }
  while((*strDest++=*strSrc++)!=0x00)
  {
    count++
  }
  return count;              //返回成功复制的字节数
}
```

再来看另一个函数memcpy()。

```
//-------------------------------------------------------------
//Name:          UINT32 memcpy(UINT8* dest,const UINT8* src,UINT32 count)
//Description:   从源地址strSrc复制数据到目的地址strDest
//               复制个数由
//Input:         desBuf -- 目的地址（从以此地址为起始位置的空间读数据）
//               srcBuf -- 源地址
//               count  -- 需要复制的数据字节数
//Output:        desBuf -- 目的地址（数据写入以此地址为起始位置的空间）
//               srcBuf -- 源地址
//Return:        >0             成功复制的数据字节数
//               ERR_PARAMETER 输入参数错误
```

```
//Remark:        注意本函数没有溢出保护，使用时候应避免count 过大产生溢出
//History:
//        1.Date:     2009/11/12
//          Author:   XXX
//          Version:  0.1
//          Modification:
//            First release version.
//-----------------------------------------------------------
UINT32 memcpy(UINT8* dest,const UINT8* src,UINT32 count)
{
  UINT32 *p1,*p2;            //用于 32bits 传送
  UINT16 *p3,*p4;            //用于 16bits 传送
  UINT8  *p5,*p6;            //用于 8bits 传送
  UINT32 size;
  if(count==0)
  {
    return ERR_PARAMETER;    //需要传输的字节数为零当作参数错误
  }
  if((UINT32)desBuf==(UINT32)srcBuf)
  {
    return ERR_PARAMETER;    //目的地址和源地址相同当作参数错误
  }
  size=count;                //得到需要传送的字节数，放在此处为零时可减少执行时间
  if((((long)desBuf&0x3)==0)&&(((long)srcBuf&0x3)==0))
  {
    //32bits mode            //目的地址和源地址都满足 4 字节对齐
    p1=(UINT32 *)desBuf;     //得到目的地址 long 指针
    p2=(UINT32 *)srcBuf;     //得到源地址 long 指针
    while(size>=4)           //还有不少于 4 字节的数据需要传送就循环
    {
      *p1=*p2;               //源地址传送 4 字节到目的地址
      size-=4;               //计数器减 4
      p1++;                  //目的地址 long 指针自加，实际上是加了 4
      p2++;                  //源地址 long 指针自加，实际上是加了 4
    }
```

```
    p5=(char *)p1;          //目的地址改用 char 指针
    p6=(char *)p2;          //源地址改用 char 指针
    while(size)             //每次循环复制 1 字节到结束
    {
      *p5=*p6;
      size--;
      p5++;
      p6++;
    }
  }
  else if((((long)desBuf&0x1)==0)&&(((long)srcBuf&0x1)==0))
  {
    //16bits mode           //目的地址和源地址都满足 2 字节对齐
    p3=(short *)desBuf;     //得到目的地址 short 指针
    p4=(short *)srcBuf;     //得到源地址 short 指针
    while(size>=2)          //还有不少于 2 字节的数据需要传送就循环
    {
      *p3=*p4;              //源地址传送 2 字节到目的地址
      size-=2;              //计数器减 2
      p1++;                 //目的地址 long 指针自加，实际上是加了 4
      p2++;                 //源地址 long 指针自加，实际上是加了 4
    }
    if(size)
    {
      (char *)p3=(char *)p4;//此时只剩余 1 字节需要复制
    }
  }
  else
  {
    //8bits mode            //目的地址或源地址不满足 2 字节对齐
    while(size)             //每次循环复制 1 字节到结束
    {
      *desBuf=*srcBuf;
      size--;
      desBuf++;
```

```
        srcBuf++;
      }
    }
    return count;              //返回实际复制的数据字节数
}
```

和 strcpy()相比 memcpy()函数会根据源地址和目的地址的状态自动选择传送方式，当源地址和目的地址均满足 4 字节对齐时，一次复制 4 个字节，当源地址和目的地址不同时满足 4 字节对齐但满足 2 字节对齐时，一次复制 2 个字节，这样做的结果是当需要复制的字节数较多而且满足一定对齐方式时复制的效率要高许多。

为什么 memcpy()函数有做对齐判断加速处理而 strcpy()函数没有呢？因为 strcpy()函数没有告诉使用者复制长度的参数 count，需要一个字节一个字节的找 0xFF 来判断结束，这样即使加入对齐判断加速处理也不会有太明显的效果，反而会使程序变得更加复杂，所以没有必要加入这些处理。

再回过头来看看开始提出的高效、安全、完善、易懂几点要求。通过 strcpy()函数已经说明我们考虑到尽可能的让程序执行效率高一些，另外函数中的循环采用 while()而不用 for()也是让代码效率高效的方式之一，通常 while()循环比 for()循环效率要高；安全性虽然这两个例子没有直接体现出来，但在注释中强调了溢出的可能性；设计了不同的返回参数就为了让程序员在使用时候更方便，这点可以归类到完善性之中；函数中的大量注释就是为了让程序易懂。

当然不能说这里设计的两个函数例子就是最好的，在编写程序时还需要根据实际情况出发决定函数该如何设计，比如这里的 memcpy()函数对于 32bits 的 MCU 加速处理是有效方法，但对一个 8bits 的 MCU 不但不能加速，反而会让效率变低。例子只是提供一个样板供大家参考，真正优质高效的函数还得要靠你们自己写出来。

函数的设计还有一点很重要，那就是函数和变量名的命名规则，最常见的是匈牙利记法，不过该记法相对比较繁琐，对于习惯汇编编程的程序员用起来会有些不习惯，另外对于一些小的单片机确实会有大材小用的感觉，所以并不建议一定按此记法来命名函数和变量。

匈牙利记法主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”，我们可以在记法的基础上根据实际情况做出一定删减或变更，定义出适合自己的命名法则，后面有自己定义的命名法则示例，这里先不细述。

4.17. 某产品函数编写规则

本文档对 V.Sxxxx Axxxxxx 的 API 函数编写规则进行约定，在编写 V.Sxxxx Axxxxxx 底层驱动函数的时候应尽量按照此文档约定进行编写。

……（详见完整版）