如何写出高效优美的单片机C语言代码

程序能跑起来并不见得你的代码就是很好的 c 代码了，衡量代码的好坏应该从以下几个方面来看

1，代码稳定，没有隐患。

2，执行效率高。

3，可读性高。

4，便于移植。

下面发一些我在网上看到的技巧和自己的一些经验来和大家分享;

1、如果可以的话少用库函数，便于不同的 mcu 和编译器间的移植

2、选择合适的算法和数据结构

应该熟悉算法语言，知道各种算法的优缺点，具体资料请参见相应的参考资料，有很多计算机书籍上都有介绍。将比较慢的顺序查找法用较快的二分查找或乱序查找法代替，插入排序或冒泡排序法用快速排序、合并排序或根排序代替，都可以大大提高程序执行的效率。.选择一种合适的数据结构也很重要，比如你在一堆随机存放的数中使用了大量的插入和删除指令，那使用链表要快得多。数组与指针语句具有十分密码的关系，一般来说，指针比较灵活简洁，而数组则比较直观,容易理解。对于大部分的编译器,使用指针比使用数组生成的代码更短，执行效率更高。但是在 Keil 中则相反,使用数组比使用的指针生成的代码更短。

3、使用尽量小的数据类型

能够使用字符型(char)定义的变量,就不要使用整型(int)变量来定义;能够使用整型变量定义的变量就不要用长整型(long int)，能不使用浮点型(float)变量就不要使用浮点型变量。当然，在定义变量后不要超过变量的作用范围，如果超过变量的范围赋值，C 编译器并不报错，但程序运行结果却错了，而且这样的错误很难发现。在 ICCAVR 中，可以在 Options 中设定使用 printf 参数，尽量使用基本型参数(%c、%d、%x、%X、%u 和%s 格式说明符)，少用长整型参数(%ld、%lu、%lx 和%lX 格式说明符)，至于浮点型的参数(%f)则尽量不要使用，其它 C 编译器也一样。在其它条件不变的情况下，使用%f 参数，会使生成的代码的数量增加很多，执行速度降低。

4、使用自加、自减指令

通常使用自加、自减指令和复合赋值表达式(如 a-=1 及 a+=1 等)都能够生成高质量的程序代码,编译器通常都能够生成 inc 和 dec 之类的指令,而使用 a=a+1 或 a=a-1 之类的指令，有很多 C 编译器都会生成二到三个字节的指令。在 AVR

单片适用的 ICCAVR、GCCAVR、IAR 等 C 编译器以上几种书写方式生成的代码是一样的，也能够生成高质量的 inc 和 dec 之类的的代码。

5、减少运算的强度

可以使用运算量小但功能相同的表达式替换原来复杂的的表达式。如下：

(1)、求余运算。

```
a=a%8;
```

可以改为：

```
a=a&7;
```

说明：位操作只需一个指令周期即可完成，而大部分的 C 编译器的“%”运算均是调用子程序来完成，代码长、执行速度慢。通常，只要求是求 2n 方的余数，均可使用位操作的方法来代替。

(2)、平方运算

```
a=pow(a,2.0);
```

可以改为：

```
a=a*a;
```

说明：在有内置硬件乘法器的单片机中(如 51 系列)，乘法运算比求平方运算快得多，因为浮点数的求平方是通过调用子程序来实现的，在自带硬件乘法器的 AVR 单片机中，如 ATMega163 中，乘法运算只需 2 个时钟周期就可以完成。既使是在没有内置硬件乘法器的 AVR 单片机中，乘法运算的子程序比平方运算的子程序代码短，执行速度快。

如果是求 3 次方，如：

```
a=pow(a,3.0);
```

更改为：

```
a=a*a*a;
```

则效率的改善更明显。

(3)、用移位实现乘除法运算

```
a=a*4;
```

b=b/4;

可以改为：

a=a<<2;

b=b>>2;

说明：通常如果需要乘以或除以 2n，都可以用移位的方法代替。在 ICCAVR 中，如果乘以 2n，都可以生成左移的代码，而乘以其它的整数或除以任何数，均调用乘除法子程序。用移位的方法得到代码比调用乘除法子程序生成的代码效率高。实际上，只要是乘以或除以一个整数，均可以用移位的方法得到结果，如：

a=a*9

可以改为：

a=(a<<3)+a

6、循环

(1)、循环语

对于一些不需要循环变量参加运算的任务可以把它们放到循环外面，这里的任务包括表达式、函数的调用、指针运算、数组访问等，应该将没有必要执行多次的操作全部集合在一起，放到一个 init 的初始化程序中进行。

(2)、延时函数：

通常使用的延时函数均采用自加的形式：

```
void delay (void)

{

unsigned int i;

for (i=0;i<1000;i++)

;

}
```

将其改为自减延时函数：

```
void delay (void)
```

```
{
unsigned int i;
for (i=1000;i>0;i--)
;
}
```

两个函数的延时效果相似，但几乎所有的 C 编译对后一种函数生成的代码均比前一种代码少 1~3 个字节，因为几乎所有的 MCU 均有为 0 转移的指令，采用后一种方式能够生成这类指令。在使用 while 循环时也一样，使用自减指令控制循环会比使用自加指令控制循环生成的代码更少 1~3 个字母。但是在循环中有通过循环变量“i”读写数组的指令时，使用预减循环时有可能使数组超界，要引起注意。

(3)while 循环和 do…while 循环

用 while 循环时有以下两种循环形式：

```
unsigned int i;
i=0;
while (i<1000)
{
i++;
//用户程序
}
```

或：

```
unsigned int i;
i=1000;
do
i--;
//用户程序
```

```
while (i>0);
```

在这两种循环中，使用 do…while 循环编译后生成的代码的长度短于 while 循环。

7、查表

在程序中一般不进行非常复杂的运算，如浮点数的乘除及开方等，以及一些复杂的数学模型的插补运算，对这些即消耗时间又消费资源的运算，应尽量使用查表的方式，并且将数据表置于程序存储区。如果直接生成所需的表比较困难，也尽量在启了，减少了程序执行过程中重复计算的工作量。

比如使用在线汇编及将字符串和一些常量保存在程序存储器中，均有利于优化

C 语言宏定义技巧(常用宏定义)

写好 C 语言，漂亮的宏定义很重要，使用宏定义可以防止出错，提高可移植性，可读性，方便性 等等。下面列举一些成熟软件中常用得宏定义。。。。。。

CODE:

1，防止一个头文件被重复包含

```
#ifndef COMDEF_H

#define COMDEF_H

//头文件内容

#endif
```

2，重新定义一些类型，防止由于各种平台和编译器的不同，而产生的类型字节数差异，方便移植。

```
typedef unsigned char boolean; /* Boolean value type. */

typedef unsigned long int uint32; /* Unsigned 32 bit value */

typedef unsigned short uint16; /* Unsigned 16 bit value */

typedef unsigned char uint8; /* Unsigned 8 bit value */

typedef signed long int int32; /* Signed 32 bit value */

typedef signed short int16; /* Signed 16 bit value */
```

```
typedef signed char int8; /* Signed 8 bit value */

//下面的不建议使用

typedef unsigned char byte; /* Unsigned 8 bit value type. */

typedef unsigned short word; /* Unsinged 16 bit value type. */

typedef unsigned long dword; /* Unsigned 32 bit value type. */

typedef unsigned char uint1; /* Unsigned 8 bit value type. */

typedef unsigned short uint2; /* Unsigned 16 bit value type. */

typedef unsigned long uint4; /* Unsigned 32 bit value type. */

typedef signed char int1; /* Signed 8 bit value type. */

typedef signed short int2; /* Signed 16 bit value type. */

typedef long int int4; /* Signed 32 bit value type. */

typedef signed long sint31; /* Signed 32 bit value */

typedef signed short sint15; /* Signed 16 bit value */

typedef signed char sint7; /* Signed 8 bit value */
```

3，得到指定地址上的一个字节或字

```
#define MEM_B( x ) ( *( (byte *) (x) ) )

#define MEM_W( x ) ( *( (word *) (x) ) )
```

4，求最大值和最小值

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

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

5，得到一个 field 在结构体(struct)中的偏移量

```
#define FPOS( type, field )

/*lint -e545 */ ( (dword) &(( type *) 0)-> field ) /*lint +e545 */
```

6,得到一个结构体中 field 所占用的字节数

```
#define FSIZ( type, field ) sizeof( ((type *) 0)->field )
```

7，按照 LSB 格式把两个字节转化为一个 Word

```
#define FLIPW( ray ) ( (((word) (ray)[0]) * 256) + (ray)[1] )
```

8，按照 LSB 格式把一个 Word 转化为两个字节

```
#define FLOPW( ray, val )
(ray)[0] = ((val) / 256);
(ray)[1] = ((val) & 0xFF)
```

9，得到一个变量的地址(word 宽度)

```
#define B_PTR( var ) ( (byte *) (void *) &(var) )
#define W_PTR( var ) ( (word *) (void *) &(var) )
```

10，得到一个字的高位和低位字节

```
#define WORD_LO(xxx) ((byte) ((word)(xxx) & 255))
#define WORD_HI(xxx) ((byte) ((word)(xxx) >> 8))
```

11，返回一个比 X 大的最接近的 8 的倍数

```
#define RND8( x ) ((((x) + 7) / 8 ) * 8 )
```

12，将一个字母转换为大写

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

13，判断字符是不是 10 进值的数字

```
#define DECCHK( c ) ((c) >= '0' && (c) <= '9')
```

14，判断字符是不是 16 进值的数字

```
#define HEXCHK( c ) ( ((c) >= '0' && (c) <= '9') ||
((c) >= 'A' && (c) <= 'F') ||
((c) >= 'a' && (c) <= 'f') )
```

15，防止溢出的一个方法

```
#define INC_SAT( val ) (val = ((val)+1 > (val)) ? (val)+1 : (val))
```

16，返回数组元素的个数

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

17，返回一个无符号数 n 尾的值 MOD_BY_POWER_OF_TWO(X,n)=X%(2^n)

```
#define MOD_BY_POWER_OF_TWO( val, mod_by )
( (dword)(val) & (dword)((mod_by)-1) )
```

18，对于 IO 空间映射在存储空间的结构，输入输出处理

```
#define inp(port) (*((volatile byte *) (port)))
#define inpw(port) (*((volatile word *) (port)))
#define inpdw(port) (*((volatile dword *)(port)))
#define outp(port, val) (*((volatile byte *) (port)) = ((byte) (val)))
#define outpw(port, val) (*((volatile word *) (port)) = ((word) (val)))
#define outpdw(port, val) (*((volatile dword *) (port)) = ((dword) (val)))
```

19,使用一些宏跟踪调试

A N S I 标准说明了五个预定义的宏名。它们是：

_ L I N E _

_ F I L E _

_ D A T E _

_ T I M E _

_ S T D C _

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序也许还提供其它预定义的宏名。

_ L I N E _及_ F I L E _宏指令在有关# l i n e的部分中已讨论，这里讨论其余的宏名。

_ D AT E _宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。

源代码翻译到目标代码的时间作为串包含在_ T I M E _中。串形式为时：分：秒。

如果实现是标准的，则宏_ S T D C _含有十进制常量1。如果它含有任何其它数，则实现是非标准的。

可以定义宏，例如：当定义了_DEBUG，输出数据信息和所在文件所在行

```
#ifdef _DEBUG

#define DEBUGMSG(msg,date)
printf(msg);printf(“%d%d%d”,date,_LINE_,_FILE_)

#else

#define DEBUGMSG(msg,date)

#endif
```

20，宏定义防止使用时错误用小括号包含。

例如：#define ADD(a,b) (a+b)

用do{}while(0)语句包含多语句防止错误

例如：#difne DO(a,b) a+b;

a++;

应用时：if(….)

DO(a,b); //产生错误

else

解决方法：#difne DO(a,b) do{a+b;

a++;}while(0)

宏中"#"和"##"的用法

一、一般用法

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.

用法:

```
#include

#include

using namespace std;

#define STR(s) #s

#define CONS(a,b) int(a##e##b)

int main()

{

printf(STR(vck)); // 输出字符串"vck"

printf("%dn", CONS(2,3)); // 2e3 输出:2000

return 0;

}
```

二、当宏参数是另一个宏的时候

需要注意的是凡宏定义里有用'#'或'##'的地方宏参数是不会再展开.

1, 非'#'和'##'的情况

```
#define TOW (2)

#define MUL(a,b) (a*b)

printf("%d*%d=%dn", TOW, TOW, MUL(TOW,TOW));
```

这行的宏会被展开为:

```
printf("%d*%d=%dn", (2), (2), ((2)*(2)));
```

MUL 里的参数 TOW 会被展开为(2).

2, 当有'#'或'##'的时候

#define A (2)

#define STR(s) #s

#define CONS(a,b) int(a##e##b)

printf("int max: %sn", STR(INT_MAX)); // INT_MAX #include

这行会被展开为：

printf("int max: %sn", "INT_MAX");

printf("%sn", CONS(A, A)); // compile error

这一行则是：

printf("%sn", int(AeA));

INT_MAX 和 A 都不会再被展开, 然而解决这个问题的方法很简单. 加多一层中间转换宏. 加这层宏的用意是把所有宏的参数在这层里全部展开, 那么在转换宏里的那一个宏(_STR)就能得到正确的宏参数.

#define A (2)

#define _STR(s) #s

#define STR(s) _STR(s) // 转换宏

#define _CONS(a,b) int(a##e##b)

#define CONS(a,b) _CONS(a,b) // 转换宏

printf("int max: %sn", STR(INT_MAX)); // INT_MAX,int 型的最大值，为一个变量 #include

输出为: int max: 0x7fffffff

STR(INT_MAX) --> _STR(0x7fffffff) 然后再转换成字符串;

printf("%dn", CONS(A, A));

输出为：200

CONS(A, A) --> _CONS((2), (2)) --> int((2)e(2))

三、'#'和'##'的一些应用特例

1、合并匿名变量名

```
#define ___ANONYMOUS1(type, var, line) type var##line

#define __ANONYMOUS0(type, line) ___ANONYMOUS1(type, _anonymous, line)

#define ANONYMOUS(type) __ANONYMOUS0(type, __LINE__)
```

例：ANONYMOUS(static int); 即：static int _anonymous70; 70 表示该行行号;

第一层：ANONYMOUS(static int); --> __ANONYMOUS0(static int, __LINE__);

第二层： --> ___ANONYMOUS1(static int, _anonymous, 70);

第三层： --> static int _anonymous70;

即每次只能解开当前层的宏，所以__LINE__在第二层才能被解开;

2、填充结构

```
#define FILL(a) {a, #a}

enum IDD{OPEN, CLOSE};

typedef struct MSG{

IDD id;

const char * msg;

}MSG;

MSG _msg[] = {FILL(OPEN), FILL(CLOSE)};
```

相当于：

```
MSG _msg[] = {{OPEN, "OPEN"},

{CLOSE, "CLOSE"}};
```

3、记录文件名

```
#define _GET_FILE_NAME(f) #f
```

```
#define GET_FILE_NAME(f) _GET_FILE_NAME(f)

static char FILE_NAME[] = GET_FILE_NAME(__FILE__);
```

4、得到一个数值类型所对应的字符串缓冲大小 #define _TYPE_BUF_SIZE(type) sizeof #type #define TYPE_BUF_SIZE(type) _TYPE_BUF_SIZE(type) char buf[TYPE_BUF_SIZE(INT_MAX)]; --> char buf[_TYPE_BUF_SIZE(0x7fffffff)]; --> char buf[sizeof "0x7fffffff"]; 这里相当于： char buf[11];