程序优化

由于单片机的性能同电脑的性能是天渊之别的，无论从空间资源上、内存资源、工作频率，都是无法

与之比较的。PC 机编程基本上不用考虑空间的占用、内存的占用的问题，最终目的就是实现功能就可以了。

对于单片机来说就截然不同了，一般的单片机的 Flash 和 Ram 的资源是以 KB 来衡量的，可想而知，单片 机的资源是少得可怜，为此我们必须想法设法榨尽其所有资源，将它的性能发挥到最佳，程序设计时必须 遵循以下几点进行优化：

1. 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。

2. 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 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。

3. 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。

（1） 求余运算

N= N %8 可以改为 N = N &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

N=Pow(3,2) 可以改为 N=3*3

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，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。

（3） 用位移代替乘法除法

N=M*8 可以改为 N=M<<3

N=M/8 可以改为 N=M>>3

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。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 N=M*9 可以改为 N=(M<<3)+M；

（4） 自加自减的区别

例如我们平时使用的延时函数都是通过采用自加的方式来实现。

```
void DelayNms(UINT16 t)
{
UINT16 i,j;
for(i=0;i<t;i++)
for(j=0;i<1000;j++)
}
```

可以改为

```
void DelayNms(UINT16 t)
{
UINT16 i,j;
for(i=t;i>=0;i--)
for(j=1000;i>=0;j--)
}
```

说明：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。

4. while 与 do...while 的区别

```
void DelayNus(UINT16 t)
{
```

```
while(t--)
{
NOP();
}
}
```

可以改为

```
void DelayNus(UINT16 t)
{
do
{
NOP();
}while(--t)
}
```

说明：使用 do…while 循环编译后生成的代码的长度短于 while 循环。

5. register 关键字

```
void UARTPrintfString(INT8 *str)
{
while(*str && str)
{
UARTSendByte(*str++)
}
```

```
}
```

可以改为

```
void UARTPrintfString(INT8 *str)
{
register INT8 *pstr=str;
while(*pstr && pstr)
{
UARTSendByte(*pstr++)
}
}
```

说明：在声明局部变量的时候可以使用 register 关键字。这就使得编译器把变量放入一个多用途的寄存器中，而不是在堆栈中，合理使用这种方法可以提高执行速度。函数调用越是频繁，越是可能提高代码的速度，注意 register 关键字只是建议编译器而已。

6. volatile 关键字

volatile 总是与优化有关，编译器有一种技术叫做数据流分析，分析程序中的变量在哪里赋值、在哪里使用、在哪里失效，分析结果可以用于常量合并，常量传播等优化，进一步可以死代码消除。一般来说,volatile 关键字只用在以下三种情况:

a) 中断服务函数中修改的供其它程序检测的变量需要加 volatile(参考本书高级实验程序)

b) 多任务环境下各任务间共享的标志应该加 volatile

c) 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义总之，volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

7. 以空间换时间

在数据校验实战当中，CRC16 循环冗余校验其实还有一种方法是查表法，通过查表可以更加快获得校验值，效率更高，当校验数据量大的时候，使用查表法优势更加明显，不过唯一的缺点是占用大量的空间。

//查表法：

```
code UINT16 szCRC16Tbl[256] = {
0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
```

0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823, 0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b, 0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12, 0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a, 0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41, 0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49, 0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70, 0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78, 0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f, 0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067, 0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e, 0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256, 0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d, 0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405, 0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c, 0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634, 0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab, 0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3, 0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a, 0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92, 0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9, 0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,

```
0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};
UINT16 CRC16CheckFromTbl(UINT8 *buf,UINT8 len)
{
UINT16 i;
UINT16 uncrcReg = 0, uncrcConst = 0xffff;
for(i = 0;i < len;i ++)
{
uncrcReg = (uncrcReg << 8) ^ szCRC16Tbl[(((uncrcConst ^ uncrcReg)
>> 8)
^ *buf++) & 0xFF];
uncrcConst <<= 8;
}
return uncrcReg;
}
```

如果系统要求实时性比较强，在 CRC16 循环冗余校验当中，推荐使用查表法，以空间换时间。

8. 宏函数取代函数

首先不推荐所有函数改为宏函数，以免出现不必要的错误。但是一些基本功能的函数很有必要使用宏函数来代替。

```
UINT8 Max(UINT8 A,UINT8 B)
```

```
{
   return (A>B?A:B)
}
```

可以改为

```
#define MAX（A，B） {(A)>(B)?(A):(B)}
```

说明：函数和宏函数的区别就在于，宏函数占用了大量的空间，而函数占用了时间。大家要知道的是，函数调用是要使用系统的栈来保存数据的，如果编译器里有栈检查选项，一般在函数的头会嵌入一些汇编语句对当前栈进行检查；同时，cpu 也要在函数调用时保存和恢复当前的现场，进行压栈和弹栈操作，所以，函数调用需要一些 cpu 时间。而宏函数不存在这个问题。宏函数仅仅作为预先写好的代码嵌入到当前程序，不会产生函数调用，所以仅仅是占用了空间，在频繁调用同一个宏函数的时候，该现象尤其突出。

9. 适当地使用算法

假如有一道算术题，求 1~100 的和。

作为程序员的我们会毫不犹豫地点击键盘写出以下的计算方法：

```
UINT16 Sum(void)
{
   UINT8 i,s;
   for(i=1;i<=100;i++)
   {
```

```
        s+=i;
    }
    return s;
}
```

很明显大家都会想到这种方法，但是效率方面并不如意，我们需要动脑筋，就是采用数学算法解决问题，
使计算效率提升一个级别。

```
UINT16 Sum(void)
{
    UINT16 s;
    s=(100 *(100+1))>>1;
    return s;
}
```

结果很明显，同样的结果不同的计算方法，运行效率会有大大不同，所以我们需要最大限度地通过数
学的方法提高程序的执行效率。

10. 用指针代替数组

在许多种情况下，可以用指针运算代替数组索引，这样做常常能产生又快又短的代码。与数组索引相
比，指针一般能使代码速度更快，占用空间更少。使用多维数组时差异更明显。下面的代码作用是相同的，

但是效率不一样。

```
UINT8 szArrayA[64];
UINT8 szArrayB[64];
UINT8 i;
UINT8 *p=szArray;
for(i=0;i<64;i++)szArrayB[i]=szArrayA[i];
for(i=0;i<64;i++)szArrayB[i]=*p++;
```

指针方法的优点是，szArrayA 的地址装入指针 p 后，在每次循环中只需对 p 增量操作。在数组索引
方法中，每次循环中都必须进行基于 i 值求数组下标的复杂运算。

11. 强制转换

C 语言精髓第一精髓就是指针的使用，第二精髓就是强制转换的使用，恰当地利用指针和强制转换不但
可以提供程序效率，而且使程序更加之简洁，由于强制转换在 C 语言编程中占有重要的地位，下面将已五
个比较典型的例子作为讲解。

例子 1：将带符号字节整型转换为无符号字节整型

```
UINT8 a=0；
INT8 b=-3；
a=(UINT8)b;
```

例子 2：在大端模式下(8051 系列单片机是大端模式)，将数组 a[2]

转化为无符号 16 位整型值。

方法 1：采用位移方法。

```
UINT8 a[2]={0x12,0x34};

UINT16 b=0;

b=(a[0]<<8)|a[1];
```

结果：b=0x1234

方法 2：强制类型转换。

```
UINT8 a[2]={0x12,0x34};

UINT16 b=0;

b= *(UINT16 *)a; //强制转换
```

结果：b=0x1234

例子 3：保存结构体数据内容。

方法 1：逐个保存。

```
typedef struct _ST

{

UINT8 a;

UINT8 b;

UINT8 c;

UINT8 d;

UINT8 e;

}ST;

ST s;
```

```
UINT8 a[5]={0};
s.a=1;
s.b=2;
s.c=3;
s.d=4;
s.e=5;
a[0]=s.a;
a[1]=s.b;
a[2]=s.c;
a[3]=s.d;
a[4]=s.e;
```

结果：数组 a 存储的内容是 1、2、3、4、5。

方法 2：强制类型转换。

```
typedef struct _ST
{
UINT8 a;
UINT8 b;
UINT8 c;
UINT8 d;
UINT8 e;
}ST;
ST s;
```

```
UINT8 a[5]={0};
UINT8 *p=(UINT8 *)&s;//强制转换
UINT8 i=0;
s.a=1;
s.b=2;
s.c=3;
s.d=4;
s.e=5;
for(i=0;i<sizeof(s);i++)
{
a[i]=*p++;
}
```

结果：数组 a 存储的内容是 1、2、3、4、5。

例子 4：在大端模式下(8051 系列单片机是大端模式)将含有位域的结构体赋给无符号字节整型值

方法 1：逐位赋值。

```
typedef struct __BYTE2BITS
{
UINT8 _bit7:1;
UINT8 _bit6:1;
UINT8 _bit5:1;
UINT8 _bit4:1;
```

```
UINT8 _bit3:1;
UINT8 _bit2:1;
UINT8 _bit1:1;
UINT8 _bit0:1;
}BYTE2BITS;
BYTE2BITS Byte2Bits;
Byte2Bits._bit7=0;
Byte2Bits._bit6=0;
Byte2Bits._bit5=1;
Byte2Bits._bit4=1;
Byte2Bits._bit3=1;
Byte2Bits._bit2=1;
Byte2Bits._bit1=0;
Byte2Bits._bit0=0;
UINT8 a=0;
a|= Byte2Bits._bit7<<7;
a|= Byte2Bits._bit6<<6;
a|= Byte2Bits._bit5<<5;
a|= Byte2Bits._bit4<<4;
a|= Byte2Bits._bit3<<3;
a|= Byte2Bits._bit2<<2;
a|= Byte2Bits._bit1<<1;
```

a|= Byte2Bits._bit0<<0;

结果：a=0x3C

方法 2：强制转换。

```
typedef struct __BYTE2BITS
{
UINT8 _bit7:1;
UINT8 _bit6:1;
UINT8 _bit5:1;
UINT8 _bit4:1;
UINT8 _bit3:1;
UINT8 _bit2:1;
UINT8 _bit1:1;
UINT8 _bit0:1;
}BYTE2BITS;
BYTE2BITS Byte2Bits;
Byte2Bits._bit7=0;
Byte2Bits._bit6=0;
Byte2Bits._bit5=1;
Byte2Bits._bit4=1;
Byte2Bits._bit3=1;
Byte2Bits._bit2=1;
Byte2Bits._bit1=0;
```

```
Byte2Bits._bit0=0;
UINT8 a=0;
a = *(UINT8 *)&Byte2Bits
```

结果：a=0x3C

例子 5：在大端模式下(8051 系列单片机是大端模式)将无符号字节整型值赋给含有位域的结构体。

方法 1：逐位赋值。

```
typedef struct __BYTE2BITS
{
UINT8 _bit7:1;
UINT8 _bit6:1;
UINT8 _bit5:1;
UINT8 _bit4:1;
UINT8 _bit3:1;
UINT8 _bit2:1;
UINT8 _bit1:1;
UINT8 _bit0:1;
}BYTE2BITS;
BYTE2BITS Byte2Bits;
UINT8 a=0x3C;
Byte2Bits._bit7=a&0x80;
Byte2Bits._bit6=a&0x40;
```

```
Byte2Bits._bit5=a&0x20;
Byte2Bits._bit4=a&0x10;
Byte2Bits._bit3=a&0x08;
Byte2Bits._bit2=a&0x04;
Byte2Bits._bit1=a&0x02;
Byte2Bits._bit0=a&0x01;
```

方法 2：强制转换。

```
typedef struct __BYTE2BITS
{
UINT8 _bit7:1;
UINT8 _bit6:1;
UINT8 _bit5:1;
UINT8 _bit4:1;
UINT8 _bit3:1;
UINT8 _bit2:1;
UINT8 _bit1:1;
UINT8 _bit0:1;
}BYTE2BITS;
BYTE2BITS Byte2Bits;
UINT8 a=0x3C;
Byte2Bits= *(BYTE2BITS *)&a;
```

12. 减少函数调用参数

使用全局变量比函数传递参数更加有效率。这样做去除了函数调用参数入栈和函数完成后参数出栈所

需要的时间。然而决定使用全局变量会影响程序的模块化和重入，故要慎重使用。

13. switch 语句中根据发生频率来进行 case 排序

switch 语句是一个普通的编程技术，编译器会产生 if-else-if 的嵌套代码，并按照顺序进行比较，

发现匹配时，就跳转到满足条件的语句执行。使用时需要注意。每一个由机器语言实现的测试和跳转仅仅

是为了决定下一步要做什么，就把宝贵的处理器时间耗尽。为了提高速度，没法把具体的情况按照它们发

生的相对频率排序。换句话说，把最可能发生的情况放在第一位，最不可能的情况放在最后。

14. 将大的 switch 语句转为嵌套 switch 语句

当 switch 语句中的 case 标号很多时，为了减少比较的次数，明智的做法是把大 switch 语句转为嵌

套 switch 语句。把发生频率高的 case 标号放在一个 switch 语句中，并且是嵌套 switch 语句的最外

层，发生相对频率相对低的 case 标号放在另一个 switch 语句中。

比如，下面的程序段把相对发生频率

低的情况放在缺省的 case 标号内。

```
UINT8 ucCurTask=1;
void Task1(void);
void Task2(void);
void Task3(void);
void Task4(void);
……
void Task16(void);
switch(ucCurTask)
{
case 1: Task1();break;
case 2: Task2();break;
case 3: Task3();break;
case 4: Task4();break;
……
case 16: Task16();break;
default:break;
}
```

可以改为

```
UINT8 ucCurTask=1;
void Task1(void);
```

```
void Task2(void);
void Task3(void);
void Task4(void);
……
void Task16(void);
switch(ucCurTask)
{
case 1: Task1();break;
case 2: Task2();break;
default:
switch(ucCurTask)
{
case 3: Task3();break;
case 4: Task4();break;
……
case 16: Task16();break;
default:break;
}
Break;
}
```

由于 switch 语句等同于 if-else-if 的嵌套代码，如果大的 if 语句同样要转换为嵌套的 if 语句。

```
UINT8 ucCurTask=1;
void Task1(void);
void Task2(void);
void Task3(void);
void Task4(void);
……
void Task16(void);
if (ucCurTask==1) Task1();
else if(ucCurTask==2) Task2();
else
{
if (ucCurTask==3) Task3();
else if(ucCurTask==4) Task4();
……
else Task16();
}
```

15. 函数指针妙用

当 switch 语句中的 case 标号很多时，或者 if 语句的比较次数过多时，为了提高程序执行速度，

可以运用函数指针来取代 switch 或 if 语句的用法,这些用法可以参考电子菜单实验代码、USB 实验代码

和网络实验代码。

```
UINT8 ucCurTask=1;
void Task1(void);
void Task2(void);
void Task3(void);
void Task4(void);
……
void Task16(void);
switch(ucCurTask)
{
case 1: Task1();break;
case 2: Task2();break;
case 3: Task3();break;
case 4: Task4();break;
……
case 16: Task16();break;
default:break;
}
```

可以改为

```
UINT8 ucCurTask=1;
void Task1(void);
void Task2(void);
```

```
void Task3(void);
void Task4(void);
……
void Task16(void);
void (*szTaskTbl)[16])(void)={Task1,Task2,Task3,Task4,…,Task16};
```

调用方法 1：(*szTaskTbl[ucCurTask])();

调用方法 2： szTaskTbl[ucCurTask]();

16. 循环嵌套

循环在编程中经常用到的，往往会出现循环嵌套。现在就已 for 循环为例。

```
UINT8 i,j;
for(i=0;i<255;i++)
{
for(j=0;j<25;j++)
{
……
}
}
```

较大的循环嵌套较小的循环编译器会浪费更加多的时间，推荐的做法就是较小的循环嵌套较大的循环。

```
UINT8 i,j;
```

```
for(j=0;j<25;j++)
{
for(i=0;i<255;i++)
{
……
}
}
```

17. 内联函数

在 C++中，关键字 inline 可以被加入到任何函数的声明中。这个关键字请求编译器用函数内部的代
码替换所有对于指出的函数的调用。这样做在两个方面快于函数调用。这样做在两个方面快于函数调用：
第一，省去了调用指令需要的执行时间；第二，省去了传递变元和传递过程需要的时间。但是使用这种方
法在优化程序速度的同时，程序长度变大了，因此需要更多的 ROM。使用这种优化在 inline 函数频繁调
用并且只包含几行代码的时候是最有效的。
如果编译器允许在 C 语言编程中能够支持 inline 关键字，注意不是 C++语言编程，而且单片机的 ROM
足够大，就可以考虑加上 inline 关键字。支持 inline 关键字的编译器如 ADS1.2，RealView MDK 等。

18. 从编译器着手

很多编译器都具有偏向于代码执行速度上的优化、代码占用空闲太小的优化。例如 Keil 开发环境编

译时可以选择偏向于代码执行速度上的优化（Favor Speed）还是代码占用空间太小的优化（Favor

Size）。还有其他基于 GCC 的开发环境一般都会提供-O0、-O1、-O2、—O3、-Os 的优化选项，而使用

-O2 的优化代码执行速度上最理想，使用-Os 优化代码占用空间大小最小。

19. 嵌入汇编---杀手锏

汇编语言是效率最高的计算机语言，在一般项目开发当中一般都采用 C 语言来开发的，因为嵌入汇编

之后会影响平台的移植性和可读性，不同平台的汇编指令是不兼容的。但是对于一些执着的程序员要求程

序获得极致的运行的效率，他们都在 C 语言中嵌入汇编，即“混合编程”。

注意：如果想嵌入汇编，一定要对汇编有深刻的了解。不到万不得已的情况，不要使用嵌入汇编。