注：本组文章主要为网友 CYPOK 的论著，由我们收集整理，特对 CYPOK 表示感谢！

1、ＰＩＣＣ和ＭＰＬＡＢ集成

ＰＩＣＣ和ＭＰＬＡＢ集成：

ＰＩＣＣ有自己的文本编辑器，不过是ＤＯＳ风格的，看来ＰＩＣＣ的工程师要专业冷到酷底了．．．

大家大可不必用它，如果你没什么癖好的话，你不会不用 UltraEdit 吧？

１：建立你的工作目录：

建议在Ｃ盘根目录下建立一个以Ａ开头的文件夹做为工作目录．因为你会发现它总是在你查找文件时候第一个跳入你眼中．

２：ＭＰＬＡＢ调用ＰＩＣＣ．（以ＭＰＬＡＢ５．７版本为例子）

启动ＭＰＬＡＢ．在 Project-->Install Language Tool:

Language Suite----->hi-tech picc

Tool Name ---->PICC Compiler

Executable ---->c:hi-pic　inpicc.exe　(假如你的ＰＩＣＣ是默认安装的)

选 Command-line

最后ＯＫ．

上面这步只需要设定一次，除非你重新安装了ＭＰＬＡＢ．

３：创建你的项目文件：（假如你实现用ＥＤＩＴ编辑好了一个叫ＡＡ．Ｃ的Ｃ代码文件）

Project-->New Project-->File Name--->myc　（假如我们把项目文件取名字叫ＭＹＣ．ＰＪＴ）

右边窗口当然要选择中你的工作目录．然后ＯＫ．

４：设定你的ＰＩＣＣ工作参数：

Project-->Edit Project

上面４个栏目就用默认的，空的也就让它空着，无所谓的．

需要修改的是：

Development Mode---->选择你的ＰＩＣ型号．当然要选择 Mplab SIM Simulator

让你可以用软件仿真．

Language Tool Suite--->HI-TECH PICC

上面的步骤，你可能会遇见多个提示条，不要管它，一路确定．

下面是ＰＩＣＣ编译器的选择项：

双击 Project Files 窗口里面的ＭＹＣ．ＨＥＸ，出现一个选择拦目．命令很多，大家可以看ＰＩＣＣ文本编辑器里面的ＨＥＬＰ，里面有详细说明．

下面就推荐几个常用也是建议用的：

Generate debug info 以及下面的２项．

Produce assembler list file

就在它们后面打勾即可，其它的不要管，除非你有特殊要求．

５：添加你的Ｃ代码文件：

当进行了前面几步后，按 Add Node 找到ＡＡ．Ｃ文件就ＯＫ了．

６：编译Ｃ代码：

最简单的一步：直接按下Ｆ１０．

编译完后，会出现各种调试信息．Ｃ代码对应的汇编代码就是工作目录里面的ＡＡ．ＩＳＴ，用ＥＤＩＴ打开可以看见详细的对比．

７：其它，要是一切都没问题，那么你就可以调试和烧片了，和以往操作无异．

2、如何从汇编转向 PICC

首先要求你要有 C 语言的基础。PICC 不支持 C++，这对于习惯了 C++的朋友还得翻翻 C 语言的书。C 代码的头文件一定要有#include<pic.h>，它是很多头文件的集合，C 编译器在 pic.h 中根据你的芯片**自动载入**相应的其它头文件。这点比汇编好用。载入的头文件中其实是声明芯片的寄存器和一些函数。顺便摘抄一个片段：

static volatile unsigned char TMR0 @ 0x01;

static volatile unsigned char PCL @ 0x02;

static volatile unsigned char STATUS @ 0x03;

可以看出和汇编的头文件中定义寄存器是差不多的。如下：

TMR0 EQU 0X01；

PCL EQU 0X02；

STATUS EQU 0X03；

都是把无聊的地址定义为大家公认的名字。

一：怎么附值？

如对 TMR0 附值，汇编中：

MOVLW 200；

MOVWF TMR0；

当然得保证当前页面在 0，不然会出错。

C 语言：

TMR0=200；//无论在任何页面都不会出错。

可以看出来 C 是很直接了当的。并且最大好处是操作一个寄存器时候，**不用考虑页面**的问题。一切由 C 自动完成。

二：怎么位操作？

汇编中的位操作是很容易的。在 C 中更简单。C 的头文件中已经对所有可能需要位操作的寄存器的每一位都有定义名称：

如：PORTA 的每一个 I/O 口定义为：RA0、RA1、RA2。。。RA7。OPTION 的每一位定义为：PS0、PS1、PS2 、PSA 、T0SE、T0CS、INTEDG 、RBPU。可以对其直接进行运算和附值。

如：

RA0=0；

RA2=1；

在汇编中是：

BCF PORTA，0；

BSF PORTA，2；

可以看出 2 者是大同小异的，只是 C 中不需要考虑页面的问题。

三：内存分配问题：

在汇编中定义一个内存是一件很小心的问题，要考虑太多的问题，稍微不注意就会出错。比如 16 位的运算等。用 C 就不需要考虑太多。下面给个例子：

16 位的除法（C 代码）：

INT X=5000；

INT Y=1000；

INT Z=X/Y；

而在汇编中则需要花太多精力。

给一个小的 C 代码，用 RA0 控制一个 LED 闪烁：

#include<pic.h>

void main()

```
{
        int x;
        CMCON=0B111;  //掉 A 口比较器，要是有比较器功能的话。
        ADCON1=0B110;       //掉 A/D 功能，要是有 A/D 功能的话。
        TRISA=0;          //RA 口全为输出。
        loop:RA0=！RA0；
        for(x=60000;--x;){;}          //延时
        goto loop;
}
```

说说 RA0=！RA0 的意思：PIC 对 PORT 寄存器操作都是先读取----修改----写入。上句的含义是程序先读 RA0，然后取反，最后把运算后的值重新写入 RA0，这就实现了闪烁的功能。

3、浅谈 PICC 的位操作

由于 PIC 处理器对位操作是最高效的，所以把一些 BOOL 变量放在一个内存的位中，既可以达到运算速度快，又可以达到最大限度节省空间的目的。在 C 中的位操作有多种选择。

****

如：char x;x=x|0B00001000; /*对 X 的 4 位置 1。*/

char x;x=x&0B11011111; /*对 X 的 5 位清 0。*/

把上面的变成公式则是：

#define bitset(var,bitno)(var |=1<<bitno)

#define bitclr(var,bitno)(var &=~(1<<bitno))

则上面的操作就是：char x;bitset(x,4)

char x;bitclr(x,5)

****

但上述的方法有缺点，就是对每一位的含义不直观，最好是能在代码中能直观看出每一位代表的意思，这样就能提高编程效率，避免出错。如果我们想用 X 的 0-2 位分别表示温度、电压、电流的 BOOL 值可以如下：

```
unsigned char x @ 0x20;                 /*象汇编那样把 X 变量定义到一个固定内存中。*/
bit temperature@ (unsigned)&x*8+0;      /*温度*/
bit voltage@ (unsigned)&x*8+1;          /*电压*/
bit current@ (unsigned)&x*8+2;          /*电流 */
```

这样定义后 X 的位就有一个形象化的名字，不再是枯燥的 1、2、3、4 等数字了。可以对 X 全局修改，也可以对每一位进行操作：

```
char=255;
temperature=0;
if(voltage).....
```

****

还有一个方法是用 C 的 struct 结构来定义：

如：

```
struct cypok{
        temperature:1;          /*温度*/
        voltage:1;              /*电压*/
        current:1;              /*电流*/
```

```
            none:4;
        }x @ 0x20;
```

这样就可以用

x.temperature=0;

if(x.current)....

等操作了。

****

上面的方法在一些简单的设计中很有效，但对于复杂的设计中就比较吃力。如象在多路工业控制上。前端需要分别收集多路的多路信号，然后再设定控制多路的多路输出。如：有 2 路控制，每一路的前端信号有温度、电压、电流。后端控制有电机、喇叭、继电器、LED。如果用汇编来实现的话，是很头疼的事情，用 C 来实现是很轻松的事情，这里也涉及到一点 C 的内存管理（其实 C 的最大优点就是内存管理）。采用如下结构：

```
union cypok{
        struct out{
                motor:1;        /*电机*/
                relay:1;        /*继电器*/
                speaker:1;      /*喇叭*/
                led1:1;         /*指示灯*/
                led2:1;         /*指示灯*/
               }out;
        struct in{
                none:5;
                temperature:1;          /*温度*/
                voltage:1;              /*电压*/
                current:1;              /*电流*/
               }in;
        char x;
};
union cypok an1;
union cypok an2;
```

上面的结构有什么好处呢？

细分了信号的路 an1 和 an2;

细分了每一路的信号的类型（是前端信号 in 还是后端信号 out):

an1.in ;

an1.out;

an2.in;

an2.out;

然后又细分了每一路信号的具体含义，如：

an1.in.temperature;

an1.out.motor;

an2.in.voltage;

an2.out.led2;等

这样的结构很直观的在 2 个内存中就表示了 2 路信号。并且可以极其方便的扩充。

如添加更多路的信号，只需要添加：

union cypok an3;

union cypok an4;

从上面就可以看出用 C 的巨大好处

4、PICC 之延时函数和循环体优化。

很多朋友说 C 中不能精确控制延时时间，不能象汇编那样直观。其实不然，对延时函数深入了解一下就能设计出一个理想的框价出来。一般的我们都用 for(x=100;--x;){;}此句等同与 x=100;while(--x){;};

或 for(x=0;x<100;x++){;}。

来写一个延时函数。

在这里要特别注意：X=100，并不表示只运行 100 个指令时间就跳出循环。

可以看看编译后的汇编：

x=100;while(--x){;}

汇编后：

```
    movlw 100
    bcf 3,5
    bcf 3,6
    movwf       _delay
  l2    decfsz      _delay
    goto l2
    return
```

从代码可以看出总的指令是是 303 个，其公式是 8+3*（X-1）。注意其中循环周期是 X-1 是 99 个。这里总结的是 x 为 char 类型的循环体，当 x 为 int 时候，其中受 X 值的影响较大。建议设计一个 char 类型的循环体，然后再用一个循环体来调用它，可以实现精确的长时间的延时。下面给出一个能精确控制延时的函数，此函数的汇编代码是最简洁、最能精确控制指令时间的：

```
void delay(char x,char y){
            char z;
        do{
                z=y;
                do{;}while(--z);
            }while(--x);
}
```

其指令时间为：7+（3*（Y-1）+7）*（X-1）如果再加上函数调用的 call 指令、页面设定、传递参数花掉的 7 个指令。则是：14+（3*（Y-1）+7）*（X-1）。如果要求不是特别严格的延时，可以用这个函数：

```
void delay(){
 unsigned int d=1000;
 while(--d){;}
}
```

此函数在 4M 晶体下产生 10003us 的延时，也就是 10MS。如果把 D 改成 2000，则是 20003us,以此类推。有朋友不明白，为什么不用 while(x--)后减量，来控制设定 X 值是多少就循环多少周期呢？现在看看编译它的汇编代码：

```
    bcf 3,5
    bcf 3,6
    movlw 10
    movwf   _delay
```

```
 l2
   decf   _delay
   incfsz   _delay,w
  goto l2
  return
```

可以看出循环体中多了一条指令，不简洁。所以在 PICC 中最好用前减量来控制循环体。

再谈谈这样的语句：

for(x=100;--x;){;}和 for(x=0;x<100;x++){;}

从字面上看 2 者意思一样，但可以通过汇编查看代码。后者代码雍长，而前者就很好的汇编出了简洁的代码。所以在 PICC 中最好用前者的形式来写循环体，好的 C 编译器会自动把增量循环化为减量循环。因为这是由处理器硬件特性决定的。PICC 并不是一个很智能的 C 编译器，所以还是人脑才是第一的，掌握一些经验对写出高效，简洁的代码是有好处的。

5、深入探讨ＰＩＣＣ之位操作

一：用位操作来做一些标志位，也就是ＢＯＯＬ变量．可以简单如下定义：

```
bit a,b,c;
```

ＰＩＣＣ会自动安排一个内存，并在此内存中自动安排一位来对应 a,b,c.由于我们只是用它们来简单的表示一些０，１信息，所以我们不需要详细的知道它们的地址＼位究竟是多少，只管拿来就用好了．

二：要是需要用一个地址固定的变量来位操作，可以参照ＰＩＣ．Ｈ里面定义寄存器．

如：用２５Ｈ内存来定义８个位变量．

```
static volatile unsigned char myvar @ 0x25;
static volatile bit b7 @ (unsigned)&myvar*8+7;
static volatile bit b6 @ (unsigned)&myvar*8+6;
static volatile bit b5 @ (unsigned)&myvar*8+5;
static volatile bit b4 @ (unsigned)&myvar*8+4;
static volatile bit b3 @ (unsigned)&myvar*8+3;
static volatile bit b2 @ (unsigned)&myvar*8+2;
static volatile bit b1 @ (unsigned)&myvar*8+1;
static volatile bit b0 @ (unsigned)&myvar*8+0;
```

这样即可以对ＭＹＶＡＲ操作，也可以对Ｂ０－－Ｂ７直接位操作．

但不好的是，此招在低档片子，如Ｃ５Ｘ系列上可能会出问题．

还有就是表达起来复杂，你不觉得输入代码受累么？呵呵

三：这也是一些常用手法：

```
#define   testbit(var, bit)    ((var) & (1 <<(bit)))          ／／测试某一位，可以做ＢＯＯＬ运算
#define   setbit(var, bit)     ((var) |= (1 << (bit)))         ／／把某一位置１
#define   clrbit(var, bit)     ((var) &= ~(1 << (bit)))          ／／把某一位清０
```

付上一段代码，可以用ＭＰＬＡＢ调试观察

```
#include<pic.h>
#define   testbit(var, bit)    ((var) & (1 <<(bit)))
#define   setbit(var, bit)     ((var) |= (1 << (bit)))
#define   clrbit(var, bit)     ((var) &= ~(1 << (bit)))
char a,b;
void main(){
 char myvar;
 myvar=0B10101010;
```

```
 a=testbit(myvar,0);
 setbit(myvar,0);
 a=testbit(myvar,0);
 clrbit(myvar,5);
 b=testbit(myvar,5);
 if(!testbit(myvar,3))
a=255;
else
a=100;
 while(1){;}
}
```

四：用标准C的共用体来表示：

```
#include<pic.h>
union var{
      unsigned char byte;
      struct {
         unsigned     b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
      } bits;
};
char a,b;
void main(){
 static union var myvar;
 myvar.byte=0B10101010;
 a=myvar.bits.b0;
 b=myvar.bits.b1;
 if(myvar.bits.b7)
  a=255;
 else
  a=100;
 while(1){;}
}
```

五：用指针转换来表示：

```
#include<pic.h>
typedef struct {
       unsigned       b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
}       bits;          //先定义一个变量的位
#define mybit0   (((bits *)&myvar)->b0)   //取 myvar 的地址（&myvar）强制转换成 bits 类型的指针
#define mybit1   (((bits *)&myvar)->b1)
#define mybit2   (((bits *)&myvar)->b2)
#define mybit3   (((bits *)&myvar)->b3)
#define mybit4   (((bits *)&myvar)->b4)
#define mybit5   (((bits *)&myvar)->b5)
#define mybit6   (((bits *)&myvar)->b6)
#define mybit7   (((bits *)&myvar)->b7)
```

```
char myvar;
char a,b;
void main(){
myvar=0B10101010;
 a=mybit0;
 b=mybit1;
 if(mybit7)
  a=255;
 else
  a=100;
 while(1){;}
}
```

六：五的方法还是烦琐，可以用粘贴符号的形式来简化它．

```
#include<pic.h>
typedef struct {
        unsigned         b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
}        bits;
#define _paste(a,b)      a##b
#define bitof(var,num)   (((bits *)&(var))->_paste(b,num))
char myvar;
char a,b;
void main(){
 a=bitof(myvar,0);
 b=bitof(myvar,1);
 if(bitof(myvar,7))
  a=255;
 else
  a=100;
 while(1){;}
}
```

有必要说说#define _paste(a,b)   a##b 的意思：

此语句是粘贴符号的意思，表示把 b 符号粘贴到 a 符号之后．

例子中是

a=bitof(myvar,0);－－－＞(((bits *)&(myvar))->_paste(b,0))－－－＞(((bits *)&(var))->b0)

可以看出来，_paste(b,0)的作用是把 0 粘贴到了 b 后面，成了 b0 符号．

总结：Ｃ语言的优势是能直接对低层硬件操作，代码可以非常非常接近汇编，上面几个例子的位操作代码是１００％的达到汇编的程度的．另一个优势是可读性高，代码灵活．上面的几个位操作方法任由你选，你不必担心会产生多余的代码量出来．

6、在 PICC 中使用常数指针。

常数指针使用非常灵活，可以给编程带来很多便利。我测试过，PICC 也支持常数指针，并且也会自动分页，实在是一大喜事。

定义一个指向 8 位 RAM 数据的常数指针（起始为 0x00):

```
#define DBYTE ((unsigned char volatile *) 0)
```

定义一个指向 16 位 RAM 数据的常数指针（起始为 0x00):

```
#define CWORD ((unsigned int volatile *) 0)
```

((unsigned char volatile *) 0)中的 0 表示指向 RAM 区域的起始地址，可以灵活修改它。

DBYTE[x]中的 x 表示偏移量。

下面是一段代码 1：

```
char a1,a2,a3,a4;
#define DBYTE ((unsigned char volatile *) 0)
void main(void){
 long cc=0x89abcdef;
 a1=DBYTE[0x24];
 a2=DBYTE[0x25];
 a3=DBYTE[0x26];
 a4=DBYTE[0x27];
 while(1);
}
```

2：

```
char a1,a2,a3,a4;
#define DBYTE ((unsigned char volatile *) 0)
void pp(char y){
 a1=DBYTE[y++];
 a2=DBYTE[y++];
 a3=DBYTE[y++];
 a4=DBYTE[y];
}

void main(void){
 long cc=0x89abcdef;
 char x;
 x=&cc;
 pp(x);

 while(1);
}
```

3：

```
char a1,a2,a3,a4;
#define DBYTE ((unsigned char volatile *) 0)
void pp(char y){
 a1=DBYTE[y++];
 a2=DBYTE[y++];
 a3=DBYTE[y++];
 a4=DBYTE[y];
}
```

```
void main(void){
 bank1 static long cc=0x89abcdef;
 char x;
 x=&cc;
 pp(x);
 while(1);
}
```

7、PICC 关于 unsigned 和 signed 的几个关键问题！

unsigned 是表示一个变量（或常数）是无符号类型。signed 表示有符号。它们表示数值范围不一样。PICC 默认所有变量都是 unsigned 类型的，哪怕你用了 signed 变量。因为有符号运算比无符号运算耗资源，而且 MCU 运算一般不涉及有符号运算。在 PICC 后面加上-SIGNED_CHAR 后缀可以告诉 PICC 把 signed 变量当作有符号处理。

在 PICC 默认的无符号运算下看这样的语句：

```
char i;
for(i=7;i>=0;i--){
;                    //中间语句
}
```

这样的 C 代码看上去是没有丁点错误的，但编译后，问题出现了：

```
      movlw 7
      movwf   i
loop
     //中间语句
     decf   i                    //只是递减，没有判断语句！！！
    goto loop
```

原因是当 i 是 0 时候，条件还成立，还得循环一次，直到 i 成负 1 条件才不成立。而 PICC 在默认参数下是不能判断负数的，所以编译过程出现问题。那么采用这样的语句来验证：

```
char i;
i=7;
while(1){
            i--;
           //中间语句
            if(i==0)break;                    //告诉 PICC 以判断 i 是否是 0 来作为条件
}
```

编译后代码正确：

```
            movlw 7
            movwf   i
loop
           //中间语句
            decfsz   i                  //判断是否是 0
             goto loop
```

再编译这样的语句：（同样循环 8 次）

```
for(i=8;i>0;i--){
```

;
}
```
movlw 8
movwf  i
loop
decfsz  i                    //同上编译的代码。
goto loop
```
再次验证了刚才的分析。
在 PICC 后面加上-SIGNED_CHAR 后缀，则第一个示例就正确编译出来了，更证明了刚才的分析是正确的。
代码如下：
```
movlw 7
movwf  i
loop
//中间语句
decf  i                  //递减
btfss  i,7               //判断 i 的 7 位来判断是否为负数
goto l94
```
总结：在 PICC 无符号编译环境下，对于递减的 for 语句的条件判断语句不能是>=0 的形式。
最后谈谈 PICC 的小窍门：
在 PICC 默认的无符号环境下，对比如下代码：
a 语句：
```
char i,j[8];
i=7;
while(1){
j[i]=0;
i--;
if(i==0)break;
}
```

b 语句：
```
char i,j[8];
for(i=8;i>0;i--){
j[i-1]=0;
}
```
表面看上去，一般会认为下面的代码编译后要大一点点，因为多了 j[i-1]中的 i-1。
其实编译后代码量是一摸一样的。
原因如下：
```
movlw   8 或 7          //a 语句是 7,b 语句是 8
movf      i
loop
//a 语句在这里提取 i 给 j 数组
//i  递减判断语句
//b 语句在这里提取 i 给 j 数组
goto loop
```

可以看出只是代码位置不同而已，并没添加代码量。b 语句同样达到了从 7 到 0 的循环。

小总结：对于递减到 0 的 for 语句推荐用>0 判断语句来实现，不会出现编译错误的问题，并且不会增加代码量，尤其对于数组操作的方面。

另：对于 PICC 或 CCS，在其默认的无符号编译环境下，如果出现负数运算就会出问题。
如(-100)+50 等，所以在编写代码时候要特别小心！！！

8、用 PICC 写高效的位移操作。

在许多模拟串行通信中需要用位移操作。
以 1-W 总线的读字节为例，原厂的代码是：

```
unsigned char read_byte(void)
{
 unsigned char i;
 unsigned char value = 0;
 for (i = 0; i < 8; i++)
 {
  if(read_bit()) value| = 0 x 01<<i;
  // reads byte in, one byte at a time and then
  // shifts it left
  delay(10); // wait for rest of timeslot
 }
 return(value);
}
```

虽然可以用，但编译后执行效率并不高效，这也是很多朋友认为 C 一定不能和汇编相比的认识提供了说法。其实完全可以深入了解 C 和汇编之间的关系，写出非常高效的 C 代码，既有 C 的便利，又有汇编的效率。首先对 for (i = 0; i < 8; i++)做手术，改成递减的形式：for(i=8;i!=0;i--),因为 CPU 判断一个数是否是 0（只需要一个指令），比判断一个数是多大来的快（需要 3 个指令）。再对 value| = 0 x 01<<i;做手术。value| = 0 x 01<<i;其实是一个低水平的代码,效率低，DALLAS 的工程师都是 NO1，奇怪为什么会如此疏忽。<I;语句其实是一个低水平的写法，效率非常低。奇怪 DALLAS 的工程师都是 NO1，怎么会如此疏忽。< P>仔细研究 C 语言的位移操作，可以发现 C 总是先把标志位清 0，然后再把此位移入字节中，也就是说，当前移动进字节的位一定是 0。那么，既然已经是 0 了，我们就只剩下一个步骤：判断总线状态是否是高来决定是否改写此位，而不需要判断总线是低的情况。于是改写如下代码：

```
for(i=8;i!=0;i--){
  value>>=1;                    //先右移一位，value 最高位一定是 0
  if(read_bit())   value|=0x80;                    //判断总线状态，如果是高，就把 value 的最高位置 1
 }
```

这样一来，整个代码变得极其高效，编译后根本就是汇编级的代码。再举一个例子：
在采集信号方面，经常是连续采集 N 次，最后求其平均值。
一般的，无论是用汇编或 C，在采集次数上都推荐用 8，16，32、64、128、256 等次数，因为这些数都比较特殊，对于 MCU 计算有很大好处。
我们以 128 次采样为例：注：sampling()为外部采样函数。

```
unsigned int total;
unsigned char i,val;
for(i=0;i<128;i++){
```

```
total+=sampling();
}
val=total/128;
```

以上代码是很多场合都可以看见的，但是效率并不怎么样，狂浪费资源。

结合 C 和汇编的关系，再加上一些技巧，就可以写出天壤之别的汇编级的 C 代码出来，首先分析 128 这个数是 0B10000000,发现其第 7 位是 1，其他低位全是 0，那么就可以判断第 7 位的状态来判断是否到了 128 次采样次数。在分析除以 128 的运算，上面的代码用了除法运算，浪费了 N 多资源，完全可以用右移的方法来代替之，val=total/128 等同于 val=(unsigned char)(total>>7);再观察下去：total>>7 还可以变通成(total<<1)>>8,先左移动一位，再右移动 8 位，不就成了右移 7 位了么？可知道位移 1，4，8 的操作只需要一个指令哦。有上面的概验了，就可以写出如下的代码：

```
unsigned int total;
unsigned char i=0
unsigned char val;
while(!(i&0x80)){              //判断 i 第 7 位，只需要一个指令。
total+=sampling();
i++;
}
val=(unsigned char)((total<<1)>>8);              //几个指令就代替了几十个指令的除法运算
```

哈哈，发现什么？代码量竟然可以减少一大半，运算速度可以提高几倍。

再回头，就可以理解为什么采样次数要用推荐的一些特殊值了。

9、C 程序优化

对程序进行优化，通常是指优化程序代码或程序执行速度。优化代码和优化速度实际上是一个予盾的统一，一般是优化了代码的尺寸，就会带来执行时间的增加，如果优化了程序的执行速度，通常会带来代码增加的副作用，很难鱼与熊掌兼得，只能在设计时掌握一个平衡点。

一、程序结构的优化

1、程序的书写结构

虽然书写格式并不会影响生成的代码质量，但是在实际编写程序时还是应该尊循一定的书写规则，一个书写清晰、明了的程序，有利于以后的维护。在书写程序时，特别是对于 While、for、do…while、if…elst、switch…case 等语句或这些语句嵌套组合时，应采用“缩格”的书写形式，

2、标识符

程序中使用的用户标识符除要遵循标识符的命名规则以外，一般不要用代数符号(如 a、b、x1、y1)作为变量名，应选取具有相关含义的英文单词(或缩写)或汉语拼音作为标识符，以增加程序的可读性，如：count、number1、red、work 等。

3、程序结构

C 语言是一种高级程序设计语言，提供了十分完备的规范化流程控制结构。因此在采用 C 语言设计单片机应用系统程序时，首先要注意尽可能采用结构化的程序设计方法，这样可使整个应用系统程序结构清晰，便于调试和维护。于一个较大的应用程序，通常将整个程序按功能分成若干个模块，不同模块完成不同的功能。各个模块可以分别编写，甚至还可以由不同的程序员编写，一般单个模块完成的功能较为简单，设计和调试也相对容易一些。在 C 语言中，一个函数就可以认为是一个模块。所谓程序模块化，不仅是要将整个程序划分成若干个功能模块，更重要的是，还应该注意保持各个模块之间变量的相对独立性，即保持模块的独立性，尽量少使用全局变量等。对于一些常用的功能模块，还可以封装为一个应用程序库，以便需要时可以直接调用。但是在使用模块化时，如果将模块分成太细太小，又会导致程序的执行效率变低(进入和退出一个函数时保护和恢复寄存器占用了一些时间)。

4、定义常数

在程序化设计过程中，对于经常使用的一些常数，如果将它直接写到程序中去，一旦常数的数值发生变化，就必须逐个找出程序中所有的常数，并逐一进行修改，这样必然会降低程序的可维护性。因此，应尽量当采用预处理命令方式来定义常数，而且还可以避免输入错误。

5、减少判断语句

能够使用条件编译(ifdef)的地方就使用条件编译而不使用 if 语句，有利于减少编译生成的代码的长度。

6、表达式

对于一个表达式中各种运算执行的优先顺序不太明确或容易混淆的地方，应当采用圆括号明确指定它们的优先顺序。一个表达式通常不能写得太复杂，如果表达式太复杂，时间久了以后，自己也不容易看得懂，不利于以后的维护。

7、函数

对于程序中的函数，在使用之前，应对函数的类型进行说明，对函数类型的说明必须保证它与原来定义的函数类型一致，对于没有参数和没有返回值类型的函数应加上“void”说明。如果果需要缩短代码的长度，可以将程序中一些公共的程序段定义为函数，在 Keil 中的高级别优化就是这样的。如果需要缩短程序的执行时间，在程序调试结束后，将部分函数用宏定义来代替。注意，应该在程序调试结束后再定义宏，因为大多数编译系统在宏展开之后才会报错，这样会增加排错的难度。

8、尽量少用全局变量，多用局部变量。因为全局变量是放在数据存储器中，定义一个全局变量，MCU 就少一个可以利用的数据存储器空间，如果定义了太多的全局变量，会导致编译器无足够的内存可以分配。而局部变量大多定位于 MCU 内部的寄存器中，在绝大多数 MCU 中，使用寄存器操作速度比数据存储器快，指令也更多更灵活，有利于生成质量更高的代码，而且局部变量所的占用的寄存器和数据存储器在不同的模块中可以重复利用。

9、设定合适的编译程序选项

许多编译程序有几种不同的优化选项，在使用前应理解各优化选项的含义，然后选用最合适的一种优化方式。通常情况下一旦选用最高级优化，编译程序会近乎病态地追求代码优化，可能会影响程序的正确性，导致程序运行出错。因此应熟悉所使用的编译器，应知道哪些参数在优化时会受到影响，哪些参数不会受到影响。

在 ICCAVR 中，有“Default”和“Enable Code Compression”两个优化选项。

在 CodeVisionAVR 中，“Tiny”和“small”两种内存模式。

在 IAR 中，共有 7 种不同的内存模式选项。

在 GCCAVR 中优化选项更多，一不小心更容易选到不恰当的选项。

二、代码的优化

1、 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。

2、 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。

3、 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。

4、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

5、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。

6、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，也尽量在启动时先计算，然后在数据存储器中生成所需的表，后以在程序运行直接查表就可以了，减少了程序执行过程中重复计算的工作量。

7、其它

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。