初谈如何从汇编转向 PICC

因为 HIDE-TECH PICC 破解版很多，所以 HIDE PICC 有比其它 PICC 有更多的用户，虽然它的编译效率不是最好。最好的是 CCS，但没破戒版。。。，不过用 HIDE PICC 精心安排函数一样可以获得很高的编译效率，还是人脑是第一的。
当然要求你要有 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裁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;   file://关掉 A 口比较器，要是有比较器功能的话。
                  ADCON1=0B110;        file://关掉 A/D 功能，要是有 A/D 功能的话。
                    TRISA=0;           file://A 口全为输出。
                   loop:RA0=！RA0；
                    for(x=60000;--x;){;}          file://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。

Pic 对位的操作

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？听小弟道来：
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。

/*bit 型变量只能是全局的或静态的，
而有时我门在实际应用中既要改变某“位"变量的值；
又要保证这个函数的独立性；那不可避免的要把
这个函数做成有参函数，可是 bit 型变量是不能用做参数的；
那该咋办泥？还好！有位段。
看看：*/
```
/*********************************************/
union FLAG
{
 unsigned char BYTE;
 struct
 {
  unsigned char b0:1;
  unsigned char b1:1;
  unsigned char b2:1;
  unsigned char b3:1;
  unsigned char b4:1;
  unsigned char b5:1;
  unsigned char b6:1;
  unsigned char b7:1;
 }bool;
};
/*********************************************/
```

```
union       FLAG              mode;
#define     auto_bit          mode.bool.b0
#define     cool_bit          mode.bool.b1
#define     dar_bit           mode.bool.b2
#define     fan_bit           mode.bool.b3
#define     heat_bit          mode.bool.b4
#define     swing_bit         mode.bool.b5
#define     bed_bit           mode.bool.b6
#define     time_bit          mode.bool.b7
/*********************************************/
void mode_task(in_mode)
union FLAG *in_mode;
{
 in_mode -> bool.b0=1;
 in_mode -> bool.b5=1;
 /*也可这样写
 in_mode -> BYTE|=0x21;*/
}
/*********************************************/
void main(void)
 {
  mode.BYTE=0X00;
  while(1)
  {
   mode_task(&mode);
  }
 }
/*********************************************/
```

这样写多爽！

这里涉及了结构，联合，位段，及指针；可得先把基础概念搞清楚！

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);
```

}

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。

数据类型和变量

算术数据类型：
单字节字符型、16 位整型、32 位长整型、24 位改进型 IEEE 标准浮点型、双精度型（24 位改进型或 32 位 IEEE 浮点型、联合类型。
集合类型：
任何数组类型、结构类型、支持位域。
其他：
指向数据和函数的指针、常数类型会被自动放在 ROM 区、可变类型。
Hi-TECH 的特殊数据类型：
用于全局或静态变量的位（布尔）类型，不能用指针进行访问，通过截断操作可以把非位操作整型转换为位操作类型，位操作尽量使用 PIC 的位处理指令，显示的位变量地趾就是本身的地址。

bit address=(byte address in which defined)*8+(offset into byte)
static near bit GIE@((unsigned)&INTCON*8)+7
例如：

```
bit sent_flag;      //definition
sent_flag=1;        //assign zero or one
```

数据类型-小结:

type	(bits)	Range
bit	1	boolean
char	8	-128 to 127
unsigned char	8	0 to 255
short	16	-32768 to 32767
unsigned short	16	0 to 65535
int	16	-32768 to 32767
unsigned int	16	0 to 65535
long	32	-2147483648 to 2147483647
unsigned long	32	0 to 4294967925
foat	24	real
double	24or32	real

char is unsigned by default.
double defaults to 24-bit.
Floating point is implemented using the IEEE754 32-bit.

存储对象种类限定符:

1.Auto 对局部变量位缺省.一个局部变量总是自动(auto)类型,除非明确声明位静态(Static)类型.

2.Static 按照 ANSI-C,静态变量在函数调用时值保持不变,除非被指针修改.

3.Extern 该存储类型表明一个变量是在另外的源文件中定义的.对于外部变量(extern)而言,本程序是不会给它分配存储空间的.

类型限定符:

1.Const 常数类型限定符通知编译器一个变量是常数而且不能被改变.任何对该变量的值的改变都会产生一条警告信息.

2.Volatile 可变类型告诉编译器一个变量在连续的操作过程中不会保持其原来的值.这就可以防止程序优化器删除表面上对可变类型的冗余引用,因为这将改变程序执行的结果.所有的输入/输出口以及那些可能被中断修改的变量都应定义为可变类型.

特殊类型限定符:

1. persistent 用于定义不需在程序启动时自动清除的变量.另外,任何持续类型(persistent)变量都被放在与其他变量不同的存储空间.

2. Bank1, Bank2,Bank3 用于 Bank1, Bank2,Bank3 区分别定义静态变量,没有 Bank 限定符意味着变量在 Bank0(Mid-range)在 Baseline 单片机中,此类限定符不影响指针变量.

3.(PIC18)Near 近类型限定符表明该类型的数据存储在直接存取区域(ACCESS BANK).近类型的对象访问无须 bank 的切换,可节省更多的空间.

绝对变量与 SFR:

Absolue 通过"construct @ address" 实现.编译器不给该变量予留

存储空间,而是仅仅让变量等于该地址.编译器和链接器不会对绝对变量和其它变量的重叠进行检查.

例如:

```
static volatile near unsigned char WDTCON @ 0XFD1;
```

这样,特殊功能寄存器(SFR'S)就可以同一般的 C 变量一样进行访问.

编译器的头文件所有 PICmicro MCU 特殊功能寄存器的予定义.<PIC.H>, <PIC18.H>分别包含了 PIC16,18 系列 MCU 各自所对应的头文件.

结构及其成员的访问:

例 1:

```
struct filtered_data{
                    char Fbandgap[4];
                    char Frefhi[4];
                    char Freflo[4];
                    char Ftemp[4];
                }Fcount;
//成员的访问通过 '.'操作符
Fcount.Fbandgap[1]=0x34;
"Fbandgap"数组的第二个元素被赋值为 0x34.
```

例 2:

```
利用例 1 中的结构变量 filtered_data:
而成员的操作用'->'操作符
声明:          struct filtered_data *ptr;
                和 ptr=&Fcount;
                ptr->Frefhi[0]=0x87;
"Frefhi"  数组的确第一个元素会被赋值为 0X87.
```

结构数组:

例:

```
struct control{
                    char mode;
                    char state;
                    char sign;
                }drive[3];
访问该数组第一个成员中的变量"mode"的值:
    drive[0].mode=0x33;
访问该数组第三个成员中的变量"sign"的值:
    drive[2].sign=0xFF;
```

联合及其成员的访问:

例 1:

```
union u_tag{
                char abc; //8bits
                int    value; //16bits
            }utemp;
```

成员的访问用'.'操作符

```
utemp.abc='A';
utemp.value=0x1234;
```

例 2:

声明: union u_tag *uptr;
和 uptr=&utemp;

成员的访问用'->'操作符

```
uptr->value=0x5678;
```

例 3:

整型变量的字节访问:

```
union{
    unsigned int var;
    struct{char var_lo;
           char var_hi;
          }hilo;
    }mix;
   char a,b;
void main(void)
   {
     mix.var=0x4321;
     a=mix.hilo.var_lo;
     b=mix.hilo.var_hi;
    }
```

结构中的位域:

例 1:

位域结构-char size

```
struct{
      unsigned int     :3; //填充位
      unsigned int b3 :1;//位 3
      unsigned int b4 :1;//位 4
      unsigned int      :3;//填充位
     }LATCbit @0xF8B;
```

下面的 C 语句用于结构中的位域"b3",其地址为 0xF8B:

```
LATCbit.b3=1;
```

注意:如果结构中的位域被分配了绝对地址(通过@construct 方式),就不会被分配存储空间.

例 2:位域结构-integer size

```
struct status{
      unsigned int high :1; //LSb
      unsigned int low    :1;
      unsigned int          :5; //填充位
      unsigned int dir    :1;
      unsigned int rate :1;
      unsigned int          :6;//填充位
      unsigned int fault:1;// MSb
```

```
}pressure;
```

下面的 C 语句用于结构变量 pressure 位域"dir":

```
pressure.dir=1;
```

注意:第一个被分配的位是整形数据的最低位. 内存会分配给该结构相应的空间.

其它的位的定义方式:

例 1:

```
union{
        unsigned char var;
        struct{
                unsigned int bit0    :1;
                unsigned int         :6;//填充位
                unsigned int         :1;
               }bits;
        }uvar;
```

现在可以:

```
uvar.var          ->前部 8 位
uvar.bits.bit7 ->只是位 7
```

例 2：

声明：static bit Flow @(unsigned)&LAB*8+4;

以下语句置位 LATB 的第 4 位“FLOW”

```
Flow=1;
```

例 3:

声明：static bit Switch @(unsigned)&PORTC*8+3;

检查 PORTC 第 3 位"Switch"的状态：

```
if(Switch){
  }
  else{
  }
```

例 4：

也可以用宏定义方式：

```
#define PortBit(port,bit)((unsigned)&(port)*8+(bit))
```

然后用下面的定义来声明任何寄存器的任位：

```
static bit led8    @PortBit(LATB,8);
static bit pulse @PortBit(LATB,7);
```

例 5：

```
bit sent_flag1;
void main(void)
{
   sent_flag=1;
   ...
   while(1);
 }
```

1-w 总线读写函数

不是自吹，这是非常高效，稳定，抗干扰的基于 PIC 的读写函数。是俺的得意之作。
在复位，写位函数中加进了检查失败操作，在读、写位函数中采用了类试串口的采 3 取 2 的采样方法，大大提高效率和可靠性。
针对汇编的 C 代码优化，大家可以对比原厂的 C 代码来比比。
喝水不忘挖井人，看试简短几个函数，可是花了偶半个月时间的心血哦。
PIC 采用 4M 晶体，所有的延时经过严格调试，修改要小心!!

```
#define false 0
#define true   1
#define uchar unsigned char
#define uint   unsigned int
#define bool   unsigned char
#define ulong unsigned long

#define Hi 1
#define Low 0
#define DQ RA4
#define SetDQ TRISA4
void delay(unsigned int val){
 while(val--);
}
bool OWReset(){
 unsigned char presence;
 unsigned char i=2;
 SetDQ=Hi;
 SetDQ=Low;                                  //设定 TRIS 来控制总线高低，这是 PIC 独特的形式
 DQ=0;                                       //刷新 PORT，避免干扰
 delay(42);  //480us
 SetDQ=Hi;
 while(--i);  //7us
 if(!DQ)
  i=1;                                       //7us 后检测总线，正常情况下总线不可能为低。
 delay(4);  //61us
 presence=DQ;
 delay(35);  //400us
 if(i || presence)
  return false;
 return true;
}
bool OWReadBit(){
 unsigned char sampling=2;
 SetDQ=Low;
```

```
 while(--sampling);
 SetDQ=Hi;
 sampling=0;
 NOP();NOP();NOP();NOP();
 if(DQ)sampling++;                          //10us 后连续 3 次采样总线
 if(DQ)sampling++;
 if(DQ)sampling++;
 delay(4);  //60us
 if(sampling>=2)                            //采 3 取 2，提高可靠性
  return 1;
 return 0;
}
unsigned char OWReadByte(){
 unsigned char i;
 unsigned char val=0;
 for(i=8;i!=0;i--){
  val>>=1;
  if(OWReadBit())
   val|=0x80;                               //最接近汇编的写法
 }
 return val;
}
bool OWWriteBit(bool bitval){
 unsigned char sampling=0;
 SetDQ=Low;
 NOP();NOP();
 if(bitval&0x01)
  SetDQ=Hi;
 NOP();NOP();NOP();NOP();
 if(DQ)sampling++;
 NOP();NOP();NOP();NOP();
 if(DQ)sampling++;
 NOP();NOP();NOP();NOP();
 if(DQ)sampling++;                          //采 3 取 2 来判断总线状态是否和写位一
致，如果有干扰或短路
                                            //可以及时检查出。
 delay(2);
 sampling=(sampling>=2);
 SetDQ=Hi;
 if(sampling==bitval)
  return true;
 return false;
}
bool OWWriteByte(unsigned char val){
```

```
 unsigned char i;
 for(i=8;i!=0;i--){
  if(!OWWriteBit(val&0x01))
   return false;
  val>>=1;
 }
 return true;
}
```

如何有效的实时控制 LED 闪烁

在很多设计中需要有精彩而实用的 LED 闪烁来表示设备工作正常与否和工作状态。
在一些实时性要求不高的设计中可以用插入延时来控制 LED 闪烁。
它的缺点现而易见：1：LED 闪烁方式反映慢。2：在延时过程不能干其它工作（中断除外），浪费了资源。3：代码雍长，真正控制 LED 就几个个指令，其它的延时代码占了 99%的空间。
如果用 TMR1 或 TMR2 来做一个时钟，上面的种种缺点就可以避免，使得你可以腾出大量的时间做更有效的工作。
下面是用 TMR1 作时钟的 C 代码(RB1、RB2、RB3 控制 LED）示例：

```
void set_tmr1(){
TMR1L=0xdc;
TMR1H=0xb;       /*设定初值 3036*/
T1CON=0B10001;           /*设定 TMR1    0.125s 溢出一次*/
}
void interrupt time(){
        if(TMR1IF){
                T1CON=0B10000;          /*关闭 TMR1*/
                TMR1L=0xdc;
                TMR1H=0xb;              /*TMR1 设初值*/
                T1CON=0B10001;          /*从新设分频比，打开 TMR1*/
                 if(s++>8){        /*每 S 清 0*/
                        s=0;
                        if(ss++>60)/*每分钟清 0*/
                                ss=0;
          }
          TMR1IF=0;
          return;
        }
}

unsigned char s;                /*每 0.125S 累加 1*/
unsigned char ss;               /*每 1 秒累加 1*/
void main(){
           set_tmr1();
.......;                        /*设定 I/O 口,开 TMR1 中断*/
```

```
        while(1){
            if(...)                    /*判断闪烁方式语句，下同*/
            RB1=(bit)(s>4);            /*每 1s 闪烁一次，占空比 50%(调节>后面值可以改变）
*/
            if(...)
            RB2=(bit)(!ss);            /*每 1 分钟闪烁一次，亮 1 秒，熄 59 秒*/
            if(...)
            RB3=(bit)(s==0 || s==2 || s== 4 || s== 6);      /*每 0.25S 闪烁一次*/
            .........;                 /*其它工作*/
          }
}
```

这样的框架对于基于要求实时性高的软件查询的程序是很有效的。

初浅研究 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。

DS1302 的 BCD 转换 C 代码

DS1302 的 C51 代码比较多，但都没提供 BCD 转换。
本人针对 DS1302 的硬件特点写了这 2 个 BCD 转换函数：
```
unsigned char bcdtodec(unsigned char bcd){//注意，BCD 低 4 位不能大于 9，最高位是控制位。
 unsigned char data;
 data=bcd&0x0f;       //取 BCD 低 4 位
 bcd=bcd&0x70;       //剔除 BCD 的最高位和低 4 位。
 data+=bcd>>1;
 data+=bcd>>3;        //用位移代替乘法运算
 return data;
```

```
}
unsigned char dectobcd(unsigned char dec){//注意 DEC 数值，比如日大于了相应月的最大日期，DS1302 将会错乱。
 unsigned char bcd;
 bcd=0;
 while(dec>=10){
  dec-=10;
  bcd++;
 }
 bcd<<=4;
 bcd|=dec;
 return bcd;
}
```

2 个函数总共才 53 字节，非常爽。
本人的门禁系统就用了它，事实证明是没有任何问题的。

使用 DS1302 一点经验：
每次单字节操作，一定要事先把 RST 和 CLK 拉低，再把 RST 抬高（也就是对 DS1302 复位）一次，不然有可能操作无效。
注意：CCS 的 DS1302 库文件就没有此步骤，实际使用有可能会出问题的，也算 CCS 的一个 BUG 吧。
对 DS1302 的年，月，日，分，秒寄存器做了写操作后，要立即读取一下（随便读某个寄存其），不然 DS1302 时钟将暂停。
对 DS1302 的寄存器做多字节写，建议在最后要对 WP 做写 0 操作，而不是写完 7 个寄存器后就停止。
DS1302 的 RST 脚要对地接 10K 电阻，当整机断电时候，此电阻可以把 RST 强行拉低，使 DS1302 的 IO 脚悬空，避免 IO 脚有可能处于输出状态而空耗后备电池。

用 PICC 写高质量的 12864 显示 C 代码

所谓高质量就是：1，高可靠。2，C 代码简洁并且编译后代码简洁。3，函数接口方便，通用性强。4，代码容易维护，修改，阅读。
本文的目的是让 PICC 新手了解 PICC 的运用，C 的技巧，C 对 PIC’MCU 的技巧，以及拿到一个新课题怎么下手等方面。
做任何课题之前，必须了解对象硬件的特性以及操作指令。
这里是 12864 的中文文挡：
一：怎么下手：
1，先读懂阅读文挡。
2，规划与 MCU 的接口脚。用 16F877A。

引脚号	引脚名称	级　别	引　　脚　　功　　能　　描　　述
1	VSS	0V	电源地
2	VDD	+5V	电源电压
3	VLCD	0～-10V	LCD 驱动负电压，要求 VDD-VLCD=13V

4	RS	H/L	读/写操作选择信号
5	R/W	H/L	寄存器选择信号
6	E	H/L	使能信号
7	DB0	H/L	八位三态并行数据总线
8	DB1		
9	DB2		
10	DB3		
11	DB4		
12	DB5		
13	DB6		
14	DB7		
15	CS1	H/L	片选信号，当 CS1=H 时,液晶左半屏显示
16	CS2	H/L	片选信号，当 CS2=H 时,液晶右半屏显示
17	/RES	H/L	复位信号,低有效
18	VEE	-10V	输出-10V 的负电压(单电源供电)
19	LED+(EL)	+5V	背光电源,Idd≤300mA
20	LED-(EL)	0V	

可以看出有 8 个数据接口 DB0-7，规划成和 877A 的 PORTD 相连。
另外有 5 条控制线，规划成和 877A 的 PORTC 相连。
于是首先写好如下的 C 代码：

```
#define CS1    RC3
#define CS2    RC4
#define RS     RC0
#define RW     RC1
#define E      RC2                    //上面是 5 条控制线的接法
#define Lcd_IO    PORTD               //数据线接 PORTD
#define SetLcd_IO   TRISD             //由于数据线是双向传输，所以要定义控制 D 口的方向
#define SetLcd_CON TRISC              //便于维护修改
```

3，MCU 执行最初总是要先设定 IO 口等寄存器。
第一个函数就是 IO 口初始化函数：

```
void ioint(){
 PORTC=0;
 PORTD=0;                    //先清 0 C、D 口，避免干扰
 SetLcd_CON =0B11100000;     //把 C 口控制线设成输出。
 SetLcd_IO=0xff;             //D 口输入。
}
```

二：对 LCD 的低层操作。
对 LCD 操作有 2 种数据，一是控制指令，二是显示数据。

指 令 名 称	控制信号		控 制 代 码							
	RS	R/W	D7	D6	D5	D4	D	D	D	D0

							3	2	1	
显示开关设置	0	0	0	0	1	1	1	1	1	D
显示起始行设置	0	0	1	1	L5	L4	L3	L2	L1	L0
页面地址设置	0	0	1	0	1	1	1	P2	P1	P0
列地址设置	0	0	0	1	C5	C4	C3	C2	C1	C0
读取状态字	0	1	BUSY	0	ON/OFF	RESET	0	0	0	0
写显示数据	1	0	数		据					
读显示数据	1	1	数		据					

并且在送数据之前要判断 LCD 是否处于忙状态。于是就先写判断 LCD 状态的函数：

```
void check_busy(){
 CS1=1;CS2=1;
 RS=0;RW=1;                    //按照上表，设定好读取 LCD 状态字的控制线。
 do{
  E=0;
  E=1;
 }while(Lcd_IO&0x80);          //判断最高位状态，如果是 1，则反复送 E 时续，直到 LCD
空闲
 CS1=0;CS2=0;
 E=0;                          //及时拉低 E，使 DB 口处于高阻状态，提高可靠性
 if(L_lcd)
  CS1=1;
 if(R_lcd)
  CS2=1;                       //通过全局 BIT 变量 L_Lcd\R_lcd 传递 LCD 左右屏幕参数
 SetLcd_IO=0;                  //使 IO 口处于输出状态，为向 LCD 写数据作准备。
}
```

后面当然就是向 LCD 写数据和指令的函数了：

```
void LCD_Write_Com(unsigned char val)
{
 check_busy();                 //检查 LCD 是否空闲
 RS=0;RW=0;                    //根据上表设定控制脚，CS1、2 在 check_busy();里事先有设定
 E=1;
 Lcd_IO=val;                   //E 时续下降沿输入数据
 E=0;
 SetLcd_IO=0xff;               //写完数据，立即把 IO 口设定成高阻，提高可靠性。
}
void LCD_Write_Dat(unsigned char val)
{
 check_busy();
 RS=1;RW=0;
```

```
 E=1;
 Lcd_IO=val;
 E=0;
 SetLcd_IO=0xff;
}
```

四：LCD 显示之前当然要对其清屏，避免出现色癍。
清屏幕就是对 LCD 内部 DRAM 全写 0。

```
void LCD_Clr(void)
{
 unsigned char i,j;
 L_lcd=1,R_lcd=1;                    //用 2 个全局 BIT 变量传递左右屏幕，因为 check_busy()
函数里面有可能
                                     //改变  CS1、CS2
// LCD_Write_Com(0x3e);              //关闭 LCD 显示，因为清屏幕极快，不要这句也罢。
 LCD_Write_Com(0xc0);
 for(j=8;j>0;j--){
  LCD_Write_Com(0xb8|j);
  LCD_Write_Com(0x40);
  for(i=64;i>0;i--){
   LCD_Write_Dat(0x00);                //对 8 个页面的 DRAM 全部送 0
  }
 }
 LCD_Write_Com(0x3f);                 //打开 LCD 显示
}
```

五：上面 4 个步骤就是 LCD 的低层操作了。下面就是显示字符的应用函数：
先要明白 LCD 的显示字符的原理，一般字符取磨都是按照 8x8,8x16,16x16 来的，分别适合小字体阿拉伯数字，大字体阿拉伯数字，汉字。
事先要把显示的字符取摸：
如：8x8 的 0-9，：

```
const char a[12][8]={
{0,62,65,65,62,0,0,0},          //0
{0,66,127,64,0,0,0,0},          //1
{0,98,81,73,70,0,0,0},
{0,34,73,73,54,0,0,0},
{0,56,38,127,32,0,0,0},
{0,79,73,73,49,0,0,0},
{0,62,73,73,50,0,0,0},
{0,3,113,9,7,0,0,0},
{0,54,73,73,54,0,0,0},
{0,38,73,73,62,0,0,0},          //9
{0,0,0,204,204,0,0,0},          //:
```

```
{0,0,0,0,0,0,0,0}                    //最后全 0，是为了对某个位置的数字清 0
};
```

显示 8x8 的函数：

```
void display8x8(uchar x,uchar y,const char *p){        //x 表示屏幕的  x 行(0-7)，y 表示屏幕的 y 列(0-15)
 uchar i;
 y<<=3;
 L_lcd=1;R_lcd=0;
 if(y&0x40){
  L_lcd=0;R_lcd=1;                                    //判断 y,来选择左右屏幕
 }
 y&=0x3f;
 LCD_Write_Com(0xb8|x);
 LCD_Write_Com(0x40|y);
 for(i=8;i>0;i--){
  LCD_Write_Dat(*p++);
 }
}
```

调用这个应用函数示例：

```
display8x8(0, 0, a[0 ]);              //在屏幕 0 行 0 列显示字符 0
display8x8(0, 15, a[9 ]);               // 0 行 15 列显示 9
display8x8(7, 8, a[10 ]);                //7 行 8 列显示：号
```

下面分别是 8x16,16x16 的函数：

```
void display8x16(unsigned char x,unsigned char y,const char *p){
 unsigned char i;
 x<<=1;
 y<<=3;
 L_lcd=1;R_lcd=0;
 if(y&0x40){
  L_lcd=0;R_lcd=1;
 }
 y&=0x3f;

       LCD_Write_Com(0xb8|x);
 LCD_Write_Com(0x40|y);
 for(i=8;i>0;i--){
  LCD_Write_Dat(*p++);
 }
 LCD_Write_Com(0xb9|x);
 LCD_Write_Com(0x40|y);
 for(i=8;i>0;i--){
  LCD_Write_Dat(*p++);
 }
```

```
}
void display16x16(uchar x,uchar y,const char *p){
 uchar i;
 x<<=1;
 y<<=4;
 L_lcd=1;R_lcd=0;
 if(y&0x40){
  L_lcd=0;R_lcd=1;
 }
 y&=0x3f;
 LCD_Write_Com(0xb8|x);
 LCD_Write_Com(0x40|y);
 for(i=16;i>0;i--){
  LCD_Write_Dat(*p++);
 }
 LCD_Write_Com(0xb9|x);
 LCD_Write_Com(0x40|y);
 for(i=16;i>0;i--){
  LCD_Write_Dat(*p++);
 }
}
```

当然 8x16 和 16x16 的字符取摸要按照规定来，存放格式雷同上面的 a 二维数组。

注意，8x16 函数只能显示 4x16 个字符，16x16 只能显示 4x8 个字符，调用函数时候，x,y 参数不能超过这个数值。

最后不要忘了在代码前加上全局 BIT 变量：

```
bit L_lcd,R_lcd;
```

此代码经过产品的长期考验，事实证明是非常可靠的。至于代码是否简洁，大家编译后就知道了。

粗浅谈谈上面代码的一些技巧地方。

```
 do{
  E=0;
  E=1;
 }while(Lcd_IO&0x80);          // Lcd_IO&0x80 意思是判断 IO 口的第 7 位，编译后只有一个位测试指令
                               //如果判断第 6 位，就是 Lcd_IO&0B01000000;以此类推。

 if(L_lcd)
  CS1=1;
 if(R_lcd)
  CS2=1;                       //为什么不用 CS1=L_lcd;CS2=R_lcd;呢？因为对于 bit 变量最好用 if 来写，编译后代码
                               //要简洁许多。
```

```
for(j=8;j>0;j--){
 for(i=64;i>0;i--){                    //许多循环体都用的是递减到 0 的形式，这样编译后代码最简洁。
  y<<=3;                               //用左移 3 位的方法来代替乘以 8，非常省空间的技巧

 if(y&0x40){                           //判断 y 是否大于 63，即测试 y 的 6 位
 y&=0x3f;                              //对 y 取 63 的摸，如果 y 是 64，则 y 取摸后成了 0。

LCD_Write_Com(0xb8|x);        //0xb8 是指令，把 x 和它相或，就可以把 x 贴入指令中。

const char a[12][8]={          //为什么要用 2 维数组，因为这样便于选择目标字符，函数接口非常直观、灵活。
                                      //用一维数组的话就费神许多，并且 2 者代码量是一样的。
          //数组用 const 修饰，是放在 ROM 中的，除非你喜欢吃 RAM，况且 PIC 没有足够的 RAM 给你

void display8x8(uchar x,uchar y,const char *p){  //调用函数，传递字符是传递字符的指针，注意 const 是指向 ROM 的。
display8x8(0, 0, a[4 ]);     //a[4]表示 2 维数组的 5 个字符的首地址，如果不太清楚标 C 的数组，那你一定要重新温习。

#define CS1    RC3
#define CS2    RC4
#define RS     RC0
#define RW     RC1
#define E      RC2                          //上面是 5 条控制线的接法
#define Lcd_IO    PORTD                     //数据线接 PORTD
#define SetLcd_IO   TRISD                   //由于数据线是双向传输，所以要定义控制 D 口的方向
#define SetLcd_CON TRISC                    //便于维护修改
```

整个代码内部没有涉及具体的 IO 口，IO 口的定义全在上面这几句里，如果你想把并口改 PORTB，就简单改上面几句就可以了，而不需要满篇代码去修改。

另外为了保证整个硬件系统运行的可靠性，一定要把 IO 口在空闲时间处理成输入形式，也就是悬空。这样一是避免偶然的干扰而烧硬件，二是降低工耗，三是。。。。，反正好处多多，百益而无一害。

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!!!

合并 2 字节的写法

很多时候需要把 2 个 CHAR 合并成 INT，把一个 INT 分解成 2 个 CHAR。
C 语言中实现这样的功能有很多写法，现在来探讨一下。
例子：把 2 个 CHAR 类型 A，B 合并成一个 INT 类型的 C。要求 B 占 C 的低 8 位，A 去掉最高位后（即 A 整体左移动 1 位后）占 C 的高 8 位。
CHIPCON 公司的 CC1020 的驱动源码是这样的，我们来看看：

```
char a,b;
int c;
c=(int) (a&0x7F)<<9 | (int) b &0x00FF;
```

用起来是没任何问题的，但是有画蛇添足之嫌。原因：
a&0x7F 是去掉最高位，但是不要忘了后面还有整体左移 9 位的运算。整体左移的运算就自动抛弃了 a 的最高位，所以 a&0x7F 可以完全不要。
后面的 b &0x00FF 也可以不要，因为 b 本来就要全部保留，并且 b 本身就没有高 8 位。
代码可以改成：

```
c=(int) (a)<<9 | (int) b;
```

可以省代码量。
但是这样的代码还是不爽，因为编译器不够智能，它不清楚<<9 可以分解成<<1 后再<<8，<<1 和<<8 这样的运算是很省代码，而且非常之快的。
于是代码又可以改成：

```
c=(((int) (a)<<1)<<8) | (int) b;
```

空间和速度有优化了很多。
但是还有最优化的方案，就是用 C 的共同体，尤其对于 A、B、C 都是局部变量的情况下。

```
union cyp{
     int INT_data;
    char CHAR_data[2];
};
union cyp val;
val.CHAR_data[0]=b;                 //b 直接赋给低 8 位
val.CHAR_data[1]=a<<1;              //a 左移动一位后直接赋给高 8 位
c=val.INT_data;
```

这样的代码是最优化的，最小的空间，最快的速度。

修改后的 CC1020 寄存器读写函数

根据 CHIPCON 公司 AN_025_source_code_1_2 源码改编（模拟 SPI 部分）。
改进之处：1：把写函数的合并字节部分去掉。
2：优化代码。
3：读写操作完后把 IO 口及时设成输入，提高可靠性。
代码经过反复测试，运行良好，测试 PASS!!

```
#define PDO         RA0
#define PDI         RA0
```

```
#define PCLK        RA7
#define PSEL        RA2
#define DIO         RA1
#define DCLK        RB0
#define SETDIO      TRISA1
#define SETPIO      TRISA0

#define INPUT   1
#define OUTPUT 0
#define FALSE 0
#define TRUE   1
//本人用 F628 的接口部分。
/*****************************************************************************/
/*   This routine writes to a single CC1020 register                         */
/*****************************************************************************/
void WriteCC1020(char val){
 char BitCounter;
 for (BitCounter=8;BitCounter!=0;BitCounter--){
  PCLK=0;
  PDI=0;
  if(val&0x80)
   PDI=1;
  val<<=1;
  PCLK=1;
 }
 PCLK=0;
}

void WriteToCC1020Register(char Address, char data)
{
 SETPIO=OUTPUT;
 PSEL=0;
 WriteCC1020((Address<<1)|0x01);       //写数最低位是 1,Address 最高位无用
 WriteCC1020(data);
 SETPIO=INPUT;
 PSEL=1;
}
/*****************************************************************************/
/*   This routine reads from a single CC1020 register                        */
/*****************************************************************************/
char ReadFromCC1020Register(char Address)
{
 char BitCounter;
 char Byte;
```

```
SETPIO=OUTPUT;
PSEL=0;
// Send address bits
WriteCC1020(Address<<1);              //读数最低位是 0，位移后最低位一定是 0。
// Set up PDATA as an input
SETPIO=INPUT;
for (BitCounter=8;BitCounter!=0;BitCounter--){
 PCLK=1;
 Byte<<=1;
 if(PDO)
  Byte|=1;
 PCLK=0;
 }
PSEL=1;
return Byte;
}
```