HT-IDE3000 Holtek C 语言

编程指南

Ver 1.1

注意

使用指南中所出现的信息在出版当时相信是正确的，然而盛群对于说明书的使用不负任何责任。文中提到的应用目的仅仅是用来做说明，盛群不保证或表示这些没有进一步修改的应用将是适当的，也不推荐它的产品使用在会由于故障或其它原因可能会对人身造成危害的地方。盛群产品不授权使用于救生、维生器件或系统中做为关键器件。盛群拥有不事先通知而修改产品的权利。对于最新的信息，请参考我们的网址 http://www.holtek.com.tw

目录

第一章

盛群 C 语言

1

简介

盛群半导体公司的 C 编译器基本上是建构于 ANSI C，由于受限于盛群单片机的硬件结构，因此只能支持部分的 ANSI C，本章节主要用来说明盛群 C 编译器所提供的 C 程序语言。

本章节包含以下的主题：

- C 语言的程序结构
- 标识符
- 数据类型
- 常量
- 运算符
- 程序流程控制
- 函数
- 指针与数组
- 结构体与共用体
- 前置处理程序伪指令
- 盛群 C 语言的扩充功能与限制

C 语言的程序结构

C 语言程序由语句、注释和前置处理程序伪指令组合而成。

语句

语句由变量、常量、运算符和函数共同组成，以分号作为结束符，并且可以执行以下的动作：

- 宣告数据变量与数据结构
- 定义数据空间
- 执行数学与逻辑运算
- 执行程序的控制动作

一行程序可以包含多个语句，复合语句由一个或多个被包含在一对大括号内的语句组成，并且可将其当成单一的语句来使用。有些语句和前置处理程序伪指令必须在盛群 C 源程序文件中使用。以下是一个整体轮廓的例子：

```
void main ()
{
/* user application source code */
}
```

源程序中必须要定义主函数 main。项目中可能会包含不只一个源程序文件，但只有一个源程序文件中可以定义主函数 main。

注释

注释经常用于在文件中解释源程序语句的意义与作用，除了在 C 关键字的中间、函数名称之间或变量名称之间外，注释可以放置在程序中的任何位置。C 编译器不对注释做处理。注释不可嵌套(nest)。盛群 C 编译器提供两种注释方式，块注释与行注释。

→ **块注释**

块注释开始于/*而结束于*/，以下为一范例：

```
/* this is a block comment */
```

块注释的结束符*/也许会与块注释的开始符/*位于不同的程序行，介于开始符/*和结束符*/ 之间所有的文字或符号都会被 C 编译器当成注释而不予编译。

→ **行注释**

行注释开始于//直到此行的结束为止。在双斜线之后的文字或符号均视为注释。范例如下：

```
// this is a line comment
```

标识符

标识符的名称包含连续的字母、数字或下划线，不过需要遵守下列规则：

- 第一个字符不可为数字
- 最长只能有 31 个字符
- 大写字母与小写字母是不同的
- 不可以使用保留字

保留字

下列为盛群 C 编译器所提供的保留字，注意要小写。

auto	bit	break	case	char
const	continue	default	do	else
enum	extern	for	goto	if
int	long	return	short	signed
static	struct	switch	typedef	union
unsigned	void	volatile	while	

盛群 C 编译器不提供 **double**、**float** 和 **register** 这三个保留字。

数据类型

数据类型与大小

盛群 C 编译器提供四种基本数据类型，分别为：

bit	单一的位
char	占用一个字节的字符
int	占用一个字节的整数
void	数值的空集合，用于函数没有返回值的类型。

接下来为可使用的限定词，分别为：

限定词	适用的数据类型	作用
const	any	将数据放入 ROM 地址区
long	int	生成一 16 位的整数
short	int	生成一 8 位的整数
signed	char, int	生成一个有符号的变量
unsigned	char, int	生成一个无符号的变量

下列为数据类型、大小与范围，分别为：

数据类型	大小	范　围
bit	1	0, 1
char	8	-128~127
unsigned char	8	0~255
int	8	-128~127
unsigned	8	0~255
short int	8	-128~127
unsigned short int	8	0~255
long	16	-32768~32767
unsigned long	16	0~65535

宣告

在定义变量的大小及数据类型之前必须先要宣告此变量的存在。宣告的语法如下：

```
data_type variable_name [,variable_name...];
```

在该范例中，*data_type* 是合法的数据类型而 *variable_name* 是变量的名称。在函数中所宣告的变量只是此函数私有的（或局部的）变量，其它函数不可以直接存取此变量。只有当函数被调用时，此函数中的局部变量才存在及有效，当执行流程从函数回到调用的程序时，局部变量便不再有效。如果变量在所有函数之外宣告，则此变量为全局变量，即所有函数均可使用、存取此变量。

限定词 **const** 可以使用在任何变量的宣告，主要是定义此变量的值为不可改变的，也就是宣告时用 **const** 限定的这个变量会存放在 ROM 地址区。限定词 **const** 也可以使用在数组变量中，**const** 变量必须在宣告时以等号和表达式设定初始值，其它的变量在宣告时不能设定初始值。

可以利用@符号宣示变量放置在某个特定的数据存储器地址，其语法如下：

```
data_type variable_name @ memory_location;
```

上例中，*memory_location* 是指定给变量的地址。如果单片机拥有多个 RAM 存储器区块，若变量要放置于编号为 0 的 RAM 存储器区块之外时，可以利用 *memory_location* 的高字节去指定所要存放的存储器区块编号。使用者可查阅盛群单片机的规格以取得可使用的 RAM 空间信息。

范例：

```
int v1 @ 0x40;   // declare v1 in the RAM bank 0 offset 0x40
int v2 @ 0x160;  // declare v2 in the RAM bank 1 offset 0x60
```

数组也可以被宣告在特定地址：

```
int port [8] @ 0x20;    // array port takes memory location
                        // 0x20 through 0x27
```

所有被盛群 C 编译器实现的变量，除了被宣告为外部变量之外，都为静态变量。无论是静态变量或是外部变量，盛群 C 编译器都不会为其预设初始值。

注意： 变量被宣告为无符号的数据类型比宣告为有符号的数据类型能够编译出效率更高的程序代码

常量

常量可以是任何数字、单一字符或字符串。

整型常量

整型常量为 int 型数据，长常量通常以 l 或 L 结尾，无符号常量则以 u 或 U 结尾，而字尾为 ul 或 UL 则表示为无符号长常量。整型常量的数值可以用下列的形式指定：

二进制常量：以 0b 或 0B 为首的数字
八进制常量：以 0 为首的数字
十六进制常量：以 0x 或 0X 为首的数字
十进制常量：非以上为首的数字

字符型常量

字符型常量是整数，它是用单引号括起来的一个字符。字符型常量的数值就是机器字符集中的字符数值。ANSI C 把转义字符(escape sequence)当作字符型常量处理。

转义字符	说明	十六进制数值
\a	警报（铃声）字符	07
\b	退格字符	08
\f	换页字符	0C
\n	换行字符	0A
\r	回车字符	0D
\t	横向跳格字符	09
\v	竖向跳格字符	0B
\\	反斜杠字符	5C
\?	问号字符	3F
\′	单引号字符	27
\″	双引号字符	22

字符串常量

字符串常量是由一对双引号括起来的零个或多个字符（包括 ANSI C 转义字符）。字符串常量是一个字符数组并且在字符的最后附加一个隐含的零值。因此，所需要的储存空间大小是双引号括起来的字符总数再加上 1。

枚举常量

整型常量的另一种命名方法称之为枚举常量，例如：

```
enum {PORTA, PORTB, PORTC};
```

定义三个整型常量的枚举常量，并且分别分配数值。

枚举常量是 int 型（-128~127），而且也可以指定一个明确的整数值给各枚举常量，例如：

```
enum {BIG=10, SMALL=20};
```

如果没有对枚举常量指定明确的数值时，第一个枚举常量值为 0，之后的枚举常量将依序加 1。枚举语句也可以被命名，例如：

```
enum boolearn {NO, YES};
```

在枚举语句中第一个名称（NO）的值为 0，下一个的名称的值是 1。

运算符

表达式是由一串运算符及操作数所组成并且指明其运算的式子，它会遵循代数的规则以计算出数值或某些负效果。表达式中某些部分计算时的顺序将会根据运算符的执行优先权和运算符所属的群组来决定。数学上常使用的运算符的结合性及交换性规则，只能应用在具有结合性和交换性的运算符。接下来讨论各种类型的运算符。

算术运算符

共有五种算术运算符。

+	加法运算符
-	减法运算符
*	乘法运算符
/	除法运算符
%	模运算符（余数为小于除数的正数或零）

模运算符%只能使用在整数的数据类型。

关系运算符

关系运算符比较两个数值，然后根据比较结果返回 TURE（真）或 FALSE（假）。

>	大于
>=	大于或等于
<	小于
<=	小于或等于

等式运算符

等式运算符类似于关系运算符。

= =	等于
!=	不等于

逻辑运算符

逻辑运算符提供 AND、OR、和 NOT 的逻辑运算并且生成 TURE（真）或 FALSE（假）值。由&&和｜｜连接的表达式由左到右计算，只要结果生成就停止计算。如果关系表达式或逻辑表达式的结果为真(TRUE)，则表达式的结果数值为 1，否则为 0。否定运算符!用来将 0 变为 1 及 1 变为 0。

&&	逻辑 AND
｜｜	逻辑 OR
!	逻辑 NOT

位运算符

提供六种运算符用于位对位的运算。位移运算符>>和<<会对运算符左边的操作数执行向右或向左的位移动，移动的位数由运算符右边的操作数指定。单操作数运算符~生成整数的 1 阶补码(oneís complement)，也就是将 1 改为 0，将 0 改为 1。

&	位 AND
｜	位 OR
^	位 XOR
~	取 1ís 补码（位反向）
>>	右移
<<	左移

复合赋值运算符

表达式的语句中总共有 10 种复合赋值运算符。对于单纯的赋值运算就是使用一个等号，以表达式计算出的数值代表等号左边的变量。另外还提供一种直接对变量本身做运算以达到修改变量的快捷方式。

<var>	+=<expr>	变量加上 expr 的值，将结果存回变量
<var>	-=<expr>	变量减去 expr 的值，将结果存回变量
<var>	*=<expr>	变量乘以 expr 的值，将结果存回变量
<var>	/=<expr>	变量除以 expr 的值，将商数存回变量
<var>	%=<expr>	变量除以 expr 的值，将余数存回变量
<var>	&=<expr>	变量与 expr 的值做位 AND 后，将结果存回变量
<var>	｜=<expr>	变量与 expr 的值做位 OR 后，将结果存回变量
<var>	^=<expr>	变量与 expr 的值做位 XOR 后，将结果存回变量
<var>	>>=<expr>	变量向右移 expr 个位后，将结果存回变量
<var>	<<=<expr>	变量向左移 expr 个位后，将结果存回变量

递增和递减运算符

递增和递减运算符可以使用在语句本身或将其插入有其它运算符的语句中。运算符的位置表示递增和递减是要在语句的计算结果之前（前缀运算符）或是之后（后缀运算符）。

++<var> 变量先加 1，再做运算
<var>++ 运算之后，变量再加 1
--<var> 变量先减 1，再做运算
<var>-- 运算之后，变量再减 1

条件运算符

条件运算符?：是一个简洁的语句，它根据表达式的结果再去执行两个语句中的一个。

<expr> ? *<statement1>* : *<statement2>*

如果*<expr>*的计算结果为一非零值（真）则*<statement1>*被执行，反之（假）则执行*<statement2>*。

逗号运算符

一组用逗号分隔的表达式，由左计算到右，而左边表达式的值会被舍弃。左边表达式的结果会先行计算出并会影响右边表达式执行的结果。整个表达式执行结果的数值和数据类型将是最右边表达式的结果数值及数据类型。

范例：

```
f (a, (t=3, t+2), c);
```

上式有三个参数，而第二个参数值为 5。

运算符的优先权与结合性

下表为运算符的优先权与结合性，优先权顺序是由高到低排列，而在同一格中的运算符拥有同等的优先权，单操作数运算符（unary operator）和复合赋值运算符的结合性为从右到左，而其它运算符的结合性为从左到右。

运算符	说明	结合性
[ ]	数组元素	由左到右
( )	小括号	
→	结构体指针	
.	结构体成员	
sizeof	数据类型的长度	
++	加 1	由右到左
--	减 1	
~	取 1 阶补码	
!	逻辑非	
-	负号	
+	正号	
&	变量地址	
*	存取指针所指地址的内容	
*	乘法运算	由左到右
/	除法运算	
%	模运算	
+	加法运算	由左到右
-	减法运算	
<<	左移运算	由左到右
>>	右移运算	
<	小于	由左到右
<=	小于或等于	
>	大于	
>=	大于或等于	
==	等于	由左到右
!=	不等于	
&	按位 AND	
^	按位 XOR	
|	按位 OR	
&&	逻辑 AND	
||	逻辑 OR	
?:	条件运算	

运算符	说明	结合性
=	赋值	由右到左
*=	相乘后存入变量	
/=	相除后存入变量	
%=	取模后存入变量	
+=	相加后存入变量	
-=	相减后存入变量	
<<=	左移后存入变量	
>>=	右移后存入变量	
&=	按位 AND 后存入变量	
|=	按位 OR 后存入变量	
^=	按位 XOR 后存入变量	
,	逗号	由左到右

类型转换

对于数据类型转换的规则而言，大都是将较小的操作数转换为较大的操作数而不致遗漏数据，例如将整数类型转换为长整数类型。从 char 转到 long 则会做正负符号的延伸。使用 cast 运算符可以将任何表达式的结果做明确的数据类型转换。例如：

```
(type-name) expression
```

expression 的结果将被转换为 *type-name* 所指定的数据类型。

程序流程控制

本节的语句都是用来控制程序执行的流程。同时也叙述如何使用控制语句中的关系与逻辑运算符以及如何执行循环。

→ **if-else** 语句

- 语法

```
if (expression)
statement1;
[else
statement2;
]
```

- 说明

if-else 是一种条件语句，语句区段的执行与否完全看 *expression* 的结果，如果 *expression* 的结果为非零值，则与其相关联的语句区段被执行，否则如果 **else** 的区段存在的话，与 **else** 相关联的语句区段就会被执行。**else** 语句与其关联的语句区段并不一定要存在。

- 范例

```
if   (word_count> 80)
{
     word_count=1;
     line++;
}
else
     word_count++;
```

→ **for** 语句

- 语法

```
for (initial-expression; condition-expression;
update-expression) statement;
```

- 说明

initial-expression 最先被执行且只执行一次，通常用来给循环的计数变量指定初始值，此变量必须在 **for** 循环之前被宣告。*condition-expression* 要在每一个循环执行前先计算，如果结果为一非零值则循环中的语句被执行，否则会跳出循环且在循环后的第一个语句将会是下一个被执行的语句。*update-expression* 会在循环内的语句执行完之后才被执行。**for** 语句可用来重复执行一行语句或一段语句。

- 范例

```
for  (i=0; i<10;i++)
     a[i]=b[i]; // copy elements from an array to another array
```

→ **while** 语句

- 语法

```
while (condition-expression)
      statement;
```

- 说明

while 语句是另一种形式的循环。当 *condition-expression* 不为零则 **while** 循环会执行 *statement*。在执行 *statement* 之前会先行查验 *condition-expression* 是否符合条件。

- 范例

```
i= 0;
while (b[i]!=0)
{
      a [i]=b[i];
      i++;
}
```

→ **do-while** 语句

- 语法

```
do
      statement;
while (condition-expression);
```

- 说明

do-while 语句是另一种形式的 **while** 循环。*statement* 会在 *condition-expression* 被计算之前先执行一次，因此在查验 *condition-expression* 之前至少会执行一次 *statement*。

- 范例

```
i=0;
do
{
      a [i]= b [i];
      i++;
} while (i<10);
```

→ **break 和 continue 语句**

- 语法

```
break;
continue;
```

- 说明

break 语句用来强迫程序立即由 **while**、**for**、**do-while** 循环和 **switch** 中跳出。**break** 语句会跳过正常的结束流程，如果它发生在嵌套循环的内部，则会返回上一层的嵌套。

Continue 语句会指示程序跳跃至循环的结束而重新开始下一轮循环。在 **while** 和 **do-while** 循环中，**Continue** 语句会强迫立即执行 *condition-expression*，而在 **for** 循环中，则会回去执行 *update-expression*。

- 范例

```
char a[10], b[10], i, j;
for (i=j=0;i<10;i++)//copy data from b[ ] to a[ ],skip blanks
{
      if (b[i]== 0) break;
      if (b[i]== 0x20) continue;
      a[j++]= b[i];
}
```

→ **goto 语句和语句标号**

- 语法

```
goto label;
```

- 说明

语句标号与变量名称的形式一样，但是其后要接冒号，其范围在整个函数中有效。

- 范例

参考 **switch** 语句的范例。

→ **switch** 语句

- 语法

```
switch (variable)
{
      case constant1:
            statement1;
            break;
      case constant2:
            statement2;
            goto Label1;
      case constant3:
            statement3;
            break;
      default:
            statement;
Label1:  statement4;
            break;
}
```

- 说明

switch 语句的 *variable* 变量用来测试变量与列表中的常量是否吻合，当吻合时此常量所属的语句被执行，并且一直执行到遇上 **break** 语句才会停止。如果 **break** 语句不存在，则程序会执行到 **switch** 程序段的结束为止。如果没有符合的常量，则执行 **default** 所属的语句，此语句并非必要的。

if-else 语句可以用来做二选一的选择，但是当有很多选择存在时就变得很麻烦了。**switch** 语句可以做多种方式的选择，当表达式的结果符合这些选择中的一个时，就跳到相关的语句执行。它相当于多个 **if-else** 语句。**switch** 语句的限制为 **switch** 变量的数据类型必须为整数，而且只能与常量值做比较。

- 范例

```
for (i=j=0;i<10;i++)
{
      switch (b[i])
      {
            case 0: goto outloop;
            case 0x20: break;
            default:
               a[j]=b[i];
               j++;
               break;
      }
}
outloop:
```

函数

在 C 语言中，所有的执行语句都必须存在于函数之内。在使用或调用函数之前，必须要定义或是宣告函数，否则 C 编译器会发出警告信息。在宣告或定义函数时，可使用两种语法，即古典形式与现代形式。针对具有多个程序存储器区块(bank)的单片机撰写程序时，函数就与变量有所不同，使用者不需要而且也没有办法将函数指定在存储器的固定区块(bank)。连接器(Linker)会将函数安排在程序存储器 ROM 的适当区块。

古典形式

```
return-type function-name (arg1, arg2, ...)
var-type arg1;
var-type arg2;
```

现代形式

```
return-type function-name (var-type arg1, var-type arg2, ...)
```

在上述两种形式中，*return-type* 是函数返回值的数据类型，如果函数没有返回值，必须将 *return-type* 宣告为 **void** 类型。*function-name* 是函数的名称，对其他所有的函数而言，它相当于一个全局变量。参数 *arg1*, *arg2* 是在此函数内使用的变量，必须指定它们的数据类型，当调用函数时，这些变量会代替形式参数储存对应的输入值。

→ 函数的宣告

```
//  classic form
    return-type function-name (arg1, arg2, ...);
//  modern form
    return-type function-name (var-type arg1, var-type arg2, ...);
```

→ 函数的定义

```
// classic form
return-type function-name (arg1, arg2, ...)
var-type arg1;
var-type arg2;
{
     statements;
}
// modern form
return-type function-name (var-type arg1, var-type arg2, ...)
{
     statements;
}
```

→ **函数参数的传递**

有两种函数参数传递的方法：

- 传值
  此方法是将参数值复制到函数中对应的形式参数。在函数中对形式参数的任何改变都不会影响到调用此函数的程序内对应变量的原始值。
- 传地址
  此方法是将参数的地址复制给函数的形式参数。在函数中，通过传入的参数地址，形式参数可以直接改变实际变量的内容，此实际变量是在调用此函数的程序内使用的。因此改变形式参数可以连带改变变量的内容。

→ **函数的返回值**

函数可以利用 **return** 语句将数值返回至调用此函数的程序。返回值必须是函数所指定的数据类型，如果 *return-type* 是 **void** 类型即表示没有返回值，应该没有数值在 **return** 语句之中。执行到 **return** 语句之后，函数会回到调用此函数的地方继续执行，任何在 **return** 语句之后的语句都不会被执行。

指针与数组

指针

指针是存有另一变量地址的变量，例如，如果一个指针变量 varpoint 存放变量 var 的地址，则 varpoint 指向 var，宣告指针变量的语法如下：

```
data-type *var_name;
```

指针的 *data-type* 需是合法的 C 数据类型，它标明了 *var_name* 所指向的变量的数据类型。在 *var_name* 之前的星号（*）是告知 C 编译器 *var_name* 为一指针变量。有两个特殊运算符（*）和（&）与指针的使用有关，例如在变量之前加上&运算符可以存取此变量的地址，而在变量之前加上*运算符则可取得此变量所指地址的内容。

除了*和&之外，还有四个运算符可以使用于指针变量，分别是+、++、- 和 . 。只有整数值才能加到指针变量或从指针变量减去。另外，当执行指针的加减运算时，指针的值会依据它所指向的数据类型的长度而调整。

数组

数组是具有相同数据类型而且可用同样名称使用的变量列表。在数组内的各变量被称为数组元素，数组的第一个元素是定义在下标为 0 的元素而最后一个元素是定义在下标为元素总数减 1 的元素。C 编译器会将一维数组(one-dimension)安置在地址连续的存储器中，第一个元素放在最小的地址。C 编译器不对数组做边界检查。

不支持将一个数组赋值给另一个数组的运算，必须从第一个数组以一次一个元素的方式复制到第二个数组对应的元素。只要是变量或常量可以使用的地方，就可以使用数组元素。

结构体与共用体(Structures and Unions)

→ 结构体

- 语法

```
struct struct-name
{
     data-type member1;
     data-type member2;
     ...
     data-type membern;
} [variable-list];
```

- 说明

结构体是一或多个相同或不相同数据类型的变量的集合，整合在单一名称下以方便处理。结构体可以被复制、赋值或传递给函数、也可由函数返回。C 编译器支持位的数据类型及嵌套式结构体。

保留字 **struct** 表示要定义一个结构体，而 *struct-name* 为此结构体的名称，在结构体中 *data-type* 必须是合法的数据类型，在结构体中的成员可以定义为不同的数据类型。*variable-list* 宣告为 *struct-name* 类型的变量，而结构体中的每一个项目是一个成员。在定义一个结构体之后，其它相同类型的变量可以使用下列语法进行宣告：

```
struct struct-name variable-list;
```

要存取结构体的成员时，必须指定变量名称及结构体的成员名称，中间加上句点分隔。语法如下：

```
variable.member1
```

variable 是结构体类型的变量而 *member1* 是结构体成员的名称。一个结构体成员的数据类型可以是前面已经定义好的结构体，这种结构即所谓的嵌套结构(nested structure)。

- 范例

```
struct person_id
{
     char id_num [6];
     char name [3];
     unsigned long birth_date;
} mark;
```

→ 共用体

- 语法

```
union union-name
{
     data-type member1;
     data-type member2;
     ...
     data-type memberm;
} [variable-list];
```

- 说明

  共用体是将不同类型的变量聚集为一群并使用相同的存储器空间。共用体类型类似于结构体类型，但是对于存储器的使用却极为不同。在结构体中所有的成员顺序地安排存储器空间，而在共用体类型中，所有的成员都从同一地址安置而且共用体类型的大小等于成员中占用最大空间的类型的大小。存取共用体类型成员的方式与存取结构体成员方式相同。

  union 是一个保留字而 *union-name* 为此共用体的名称，*variable-list* 定义有相同数据类型的变量，可有可无。

- 范例

```
union common_area
{
     char name [ 3 ];
     int id;
     long data;
} cdata;
```

前置处理伪指令

前置处理伪指令将会指导编译器如何去编译源程序代码。此伪指令类似一个简单的宏处理器，在编译器正式编译源程序前先行处理某些程序。一般而言，前置处理伪指令不会直接编译成执行码。C 编译器在编译的初期会将源程序中的前置处理指令行移除并做适当的处理，同时也会将调用宏指令的程序替换为宏展开后的程序，以及其它的数据，例如**#line** 命令。前置处理伪指令以#号做为开头，即以#号开头的程序行即被视为前置处理伪指令，其后则为命令的名称，以下为前置处理伪指令：

→ 宏替换：**#define**

- 语法

  #define *name replaced-text*
  #define *name* [(*parameter-list*)] *replaced-text*

- 说明

  #define 伪指令定义字符串常量。在编译源程序之前，会以定义的字符串常量替换到程序中。其主要目的是增加源程序的可读性与维护性。如果无法在一行中写完 *replaced-text*，可以使用反斜线(\)表示还有更多的程序行。

- 范例

```
#define TOTAL_COUNT   40
#define USERNAME      Henry
#define MAX(a,b)      (((a)>(b))?(a):(b))
#define SWAP(a,b)     {int tmp;\
                       tmp=a;\
                       b=a;\
                       a=tmp;}
```

→ **#error**

- 语法

  #error "*message-string*"

- 说明

  #error 伪指令会生成一个使用者所定义的诊断信息，*message-string*。

- 范例

```
#if         TOTAL_COUNT > 100
#error      "Too many count."
#endif
```

→ 条件编译：**#if #else #endif**

- 语法

  #if *expression*
  source codes1
  [**#else**
  source codes2]
  #endif

- 说明

  #if 和**#endif** 是一组用来做条件编译程序的伪指令，而编译条件则取决于 *expression* 的运算值。**#else** 伪指令提供二选一的编译方式，它是可有可无的。如果 *expression* 运算的结果不为零，则 *source codes1* 将被编译，否则如果有 *source codes2*，则 *source code2* 被编译。

- 范例

```
#define MODE 2
#if MODE > 0
     #define DISP_MODE MODE
#else
     #define DISP_MODE 7
#endif
```

→ **条件编译：#ifdef**

- 语法

#ifdef *symbol*
source codes1
[**#ifdef**
source codes2]
#endif

- 说明

#ifdef 伪指令类似**#if** 伪指令，但是它不是以表达式的结果决定编译的程序行，而是以检查所指定的 *symbol* 是否已经被定义的方式决定的。**#else** 伪指令提供二选一的编译方式，它不是一定要有的。如果 *symbol* 已经被定义则 *source codes1* 将被编译，否则如果 *source codes2* 存在，它将被编译。

- 范例

```
#ifdef DEBUG_MODE
#define TOTAL_COUNT 100
#endif
```

→ **条件编译：#ifndef**

- 语法

#ifndef *symbol*
source codes1
[**#else**
source codes2]
#endif

- 说明

#ifndef 伪指令与**#ifdef** 伪指令类似。**#else** 伪指令提供二选一的编译方式，它是可有可无的。如果 *symbol* 没有被定义则 *source codes1* 将被编译，否则如果 *source codes2* 存在，它将被编译。

- 范例

```
#ifndef DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

→ 条件编译：**#elif**

- 语法

```
#if  expression1
     source codes1
#elif     expression2
     source codes2
[#else
     source codes3]
#endif
```

- 说明

#elif 伪指令是要随**#if** 伪指令一起使用。除了通常使用的两种条件编译外，它提供第三种条件编译方式。如果 *expression1* 为非零，则 *source codes1* 被编译，如果 *expression1* 为零，则 *expression2* 被检查是否为非零，如果是，则 *source codes2* 被编译，否则，如果 *source codes3* 存在，则 *source codes3* 被编译。

- 范例

```
#if MODE==1
#define DISP_MODE 1
#elif MODE==2
#define DISP_MODE 7
#endif
```

→ 条件编译：**defined**

- 语法

```
#if defined symbol
     source code1
[#else
     source code2
#endif
```

- 说明

单操作数的运算符 **defined** 可以使用在**#if** 或**#elif** 伪指令中。如下列格式的流程控制行 #ifdef symbol 则等于#if defined symbol 的效果，而#ifndef symbol 则等于#if !defined symbol 的效果。

- 范例

```
#if defined DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

→ 条件编译：**#undef**

- 语法

  #undef *symbol*

- 说明

  #undef 伪指令会将先前已定义的符号清除，相当于此符号没有被定义。符号一旦被定义，就会一直保持在定义的状态中，直到编译程序的结尾或者使用**#undef** 伪指令取消定义。

- 范例

```
#define TOTAL_COUNT 100
...
#undef TOTAL_COUNT
#define TOTAL_COUNT 50
```

→ 文件包含：**#include**

- 语法

  #include <*file-name*>
  or
  #include "*file-name*"

- 说明

  #include 会将指定文件的内容插入到源程序文件中。当使用<*file-name*>的格式时，编译器会从环境变量 INCLUDE 所指定的路径中寻找 *file-name* 文件，如果没有定义 INCLUDE，C 编译器会在指定的路径中搜寻文件。如果使用"*file-name*"的格式，则 C 编译器会以指定的方式搜寻 *file-name* 文件，如果没有指定路径，则会从当前所在的路径中找寻文件。

- 范例

```
#include <ht48c10-1.h>
#include "my.h"
```

盛群 C 语言的扩充功能与限制

盛群 C 语言提供 ANSIC 以外的许多扩充功能，大部分是供给盛群公司各类型单片机使用的。但是由于单片机的资源有限，必须遵守某些限制。

关键字

以下为盛群 C 语言中可使用的关键字：

@ bit norambank rambank0 vector

下面的关键字与限定词是不能使用的：

double float Register

存储器区块(memory bank)

对于地址在较高的储存区块（非区块 0）的变量而言，必须使用间接寻址模式去存取它，程序编译后的指令数及执行效果比较不佳。为了达成更大的效益，可以将程序中经常使用到的变量定义在数据存储器的储存区块 0。盛群 C 语言提供关键字 **rambank0** 用来宣告变量在储存区块 0。

- 语法

```
#pragma rambank0
//data declarations
#pragma norambank
```

- 说明

  rambank0 会指示盛群 C 编译器将其后的变量定位于储存区块 0 中，直到出现 **norambank** 关键字或是到程序的结束。对于只有单一数据存储器区块的单片机，这两个关键字则无效。

- 范例

```
#pragma rambank0
unsigned int i, j;      // 变量 i, j 放在储存区块 0
long len;               // 变量 len 放在储存区块 0

#pragma norambank
unsigned int iflag      // 变量 iflag 的储存区块不确定
                        // 因为 norambank
#pragma rambank0
int tmp;                // 变量 tmp 放在储存区块 0
...
i=1;                    //MOV A, 1  (编译后的汇编语言指令)
                        //MOV _i, A

iflag=1                 //MOV A, BANK_iflag
                        //MOV [04H], A
                        //MOV A, OFFSET_iflag
                        //MOV [03H], A
                        //MOV A, 1
                        //MOV [02H], A
```

位数据类型

盛群 C 语言提供位数据类型，此类型可用于变量的宣告、参数列表以及函数的返回值。位变量的宣告与其它 C 数据类型的变量宣告一样。对于具有多 RAM/ROM 储存区块的单片机，应该将位变量宣告在 RAM 储存区块 0（**#pragma rambank0**）。

- 范例

```
#pragma rambank0
bit test_flag;              // 位变量应该放在储存区块 0

bit testfunc(               //bit function
bit f1,                     //bit arguments
bit f2)
{
     ...
     return 0;              //return bit value
}
```

- 限制
  - 为了利用位数据类型的优点，不建议将变量宣告为位数组的数据类型。
  - 指针不可设定为位类型。

内嵌式汇编语言

- 语法

```
#asm
[label:] opcode [operands]
...
#endasm
```

- 说明

  #asm 和**#endasm** 是内嵌式汇编语言的前置处理程序伪指令。C 编译器会将 **#asm** 之后的(或夹在**#asm** 和**#endasm** 之间的)汇编语言指令直接写进输出的文件，有如直接使用汇编语言撰写程序。

- 范例

```
//convert low nibble value in the accumulator to ASCII
#asm
; this is an inline assembly comment
and a, 0fh
sub a, 09h
sz c add a, 40h-30h-9
add a, 30h+9
#endasm
```

中断

盛群 C 语言提供一种使用伪指令**#pragma** 以实现中断服务程序的方法（ISRs）。伪指令**#pragma vector** 用来宣告 ISR 的名称与存储器地址，之后若有函数的名称与**#pragma vector** 定义的符号名称相同时，此函数就是这个中断向量的中断服务程序。在中断服务程序中的 return 语句将会编译成**RETI** 指令。

- 语法

  #pragma vector *symbol @ address*

- 说明

  symbol 是中断服务程序的名称，*address* 是中断地址，复位向量（地址 0）固定由主函数 main( )使用，任何中断服务程序不可使用此中断地址。

- 限制

  撰写 ISR 程序时要注意有四种限制：

  – ISR 没有输入参数且返回类型是 **void**。
  – ISR 不能够重复进入，且而在 ISR 中不可让任何中断再发生。
  – 在程序中不要直接调用 ISR 程序，应该由中断信号输入时自行调用它。
  – 在 ISR 中不要调用用 C 写的函数。但是可以调用系统函数或 C 编译器内建的函数(built-in function) 。如果必须在 ISR 中调用函数，可以使用汇编语言撰写这个函数。

- 范例

```
#pragma vector timer0 @ 0x8
extern void ASM_FUNCTION();
void setbusy(){
...
}

void timer0(){
      ...
      ASM_FUNCTION();     //The ASM_FUNCTION should be an
                          //assembly function

      _delay(3)           //Ok; built-in function

      setbusy();          //Wrong! Do not call function
}
```

变量

运算符“@”用来指定数据存储器中变量的地址。

- 语法

```
data_type variable_name @ memory_location
```

- 说明

memory_location 指定变量所在的地址。在只有单一 RAM/ROM 存储器区块的单片机中，*memory_location* 是一个字节，而在具有多个 RAM/ROM 存储器区块的单片机中，*memory_location* 是两个字节，高字节存放存储器区块的编号。请参考盛群单片机的资料手册以取得 RAM 存储器空间的信息。

- 范例

```
int v1 @ 0x5B;     // 宣告变量 v1 放置于 RAM bank 0, offset 0x5B
int v2 @ 0x2F0;    // 宣告变量 v2 放置于 RAM bank 2, offset 0xF0
```

静态变量

盛群 C 语言提供有效范围在文件内的静态变量，而不支持局部的静态变量。

- 范例

```
static i;               // 宣告静态变量, 以文件为有效范围
void  f1 () {
      i=1;              // OK 可以使用此变量
}
void f2 () {
      static int j;     // 错误的宣告,不能将函数中的局部变量宣告为静
                           态变量
                        // local static variable is not supported
      ...
}
```

常量

盛群 C 语言支持二进制常量，任何以 0b 或 0B 开头的字符串将被视为二进制常量。

- 范例

```
0b101=5
0b1110=14
```

函数

避免撰写重复进入与递归的程序行。

数组

任何数组应该配置在一个连续的存储器区内，而且最大不可超过 256 个元素。严格来说，数组的大小取决于所使用单片机的数据存储器区的大小。

常量

常量必须宣告为全局型且在宣告时就要设定初始值。常量不可宣告为外部使用。数组常量需要指定数组的大小，否则会产生错误。

```
const char carray [ ]={1,2,3}; // 错误,没有指定数组的大小
const char carray [3]={1,2,3}; // 正确
```

字符串常量必须在包含 main( ) 主函数的 C 语言文件中使用。

```
//test.c
void f1 (char *s);
void f2 () {
     f1 ("abcd")    //"abcd" 是字符串常量
                    // 如果在文件 test.c 中没有定义 main( )主函数
                    // 则 Holtek C 编译器会发出错误信息
     ...
}
...
void main(){
     ...
}
```

指针

指针不能用于常量与位变量。

初始值

全局变量宣告时不可以同时设定初始值，局部变量则无此项限制，但是常量在宣告时则一定要设定初始值。

- 范例

```
unsigned int i1=0;        // 错误; 全局变量, 不可设定初始值

unsigned int i2;          // 正确
const unsigned int i3;    // 错误; 常量, 必须设定初始值

const unsigned int i4=5; // 正确
const char a1[5];         // 错误; 数组常量, 必须设定初始值

const char a2[5]={0x1 0x2 0x3 0x4 0x5};  // 正确
const char a2[4]="abc";  //={'a', 'b', 'c', 0}
const char a2[3]="abc";  //={'a', 'b', 'c'}
const char a2[2]="abc";  // 数组大小不一致
```

乘法/除法/模

乘法、除法和模（“*”，“/”，“%”）运算符由系统调用执行。

内建函数

- WDT & halt & nop

C 系统调用	汇编语言码
void_clrwdt ( )	CLR WDT
void_clrwdt1 ( )	CLR WDT1
void_clrwdt2 ( )	CLR WDT2
void_halt ( )	HALT
void_nop ( )	NOP

- 左移/右移

```
void_rr (int*);       //rotate 8 bits data right
void_rrc (int*);      //rotate 8 bits data right through carry
void_lrr (long*);     //rotate 16 bits data right
void_lrrc (long*);    //rotate 16 bits data right through carry
void_rl (int*);       //rotate 8 bits data left
void_rlc (int*);      //rotate 8 bits data left through carry
void_lrl (long*);     //rotate 16 bits data left
void_lrlc (long*);    //rotate 16 bits data left through carry
```

- 高/低半字节的交换

```
void_swap (int*);     //swap nibbles of 8 bits data
```

- 以指令周期为单位的延迟函数

```
void_delay(unsigned long);   //delay n instruction cycle
```

_delay 函数强迫单片机去执行所指定的周期数。周期数为零则会执行无穷的循环。

_delay 函数的参数只能为常量值并不接受变量。

- 范例 1

```
// 假设 watch dog timer 看门狗定时器已经启动
// 看门狗定时器的清除指令选择为使用一条清除指令
void error (){
      delay (0);            // 无穷的循环, 类似 while(1);
}
void dotest(){
      unsigned int ui;
      ui =0x1;
      rr(&ui);              //rotate right
      if (ui != (unsigned int)0x80) error();
      ui =0xab;
      swap(&ui);
      if (ui != (unsigned int)0xba) error();
}
```

```
void main(){
        unsigned int i;
        for(i=0; i<100; i++){
            _clrwdt();
            _delay(10);    // 延迟 10 个指令周期
            dotest();
        }
    }
```

- 范例 2

```
// 假设 watch dog timer 看门狗定时器已经启动
// 看门狗定时器的清除指令选择为使用两条清除指令
void do test(){
...
}
void main(){
    unsigned int i;
    for(i=0; i<100; i++){
        _clrwdt1();
        _clrwdt2();
        dotest();
    }
}
```

堆栈

因为盛群单片机堆栈的层数是有限的，所以要考虑函数调用时的层数以避免堆栈溢出。乘法、除法、取模和常量是使用“call”指令实现其功能的，都只占用一层的堆栈。

运算符/系统函数	所需要的堆栈层数
main()	0
_clrwdt()	0
_clrwdt1()	0
_clrwdt2()	0
_halt()	0
_nop()	0
_rr(int*);	0
_rrc(int*);	0
_lrr(long*);	0
_lrrc(long*);	0
_rl(int*);	0
_rlc(int*);	0
_lrl(long*);	0
_lrlc(long*);	0
_delay(unsigned long)	1
*	1
/	1
%	1
constant array	1

第二章

混合语言 2

盛群程序工具组（汇编编译器、连接器、函数库管理器和盛群 C 编译器）为混合语言的编程提供了一些方法，如使用盛群的汇编语言和 C 语言。也就是说，项目中的源程序文件可以同时使用汇编语言和 C 语言去完成。然而，程序设计师在使用这两种语言设计程序时应该遵守一些规则。为了帮助程序的顺利完成，本章将说明盛群 C 编译器在将 C 语言程序译成汇编程序时所遵循的一些常规惯例以及如何定义子程序的名称等。以下即为相关的议题：

- Little endian
- 函数与参数的命名规则
- 参数的传递
- 返回值
- 寄存器内容的保存
- 在 C 程序中调用汇编语言函数
- 在汇编程序中调用 C 函数
- 使用汇编语言编写 ISR 函数

Little Endian

盛群 C 编译器采用 little-Endian 的数据格式，即一个 WORD 的低字节是此 WORD 的最低的字节(least significant byte)，而高字节则是最高的字节(most significant byte)，在存储器的配置中，低字节占用较低的地址，而高字节占用较高的地址。

范例

```
long var @ 0x40;
var = 0x1234;
```

地址 0x40 存放数据 0x34，地址 0x41 存放数据 0x12。

函数与参数的命名规则

盛群的汇编编译器（Assembler）在处理符号名称时是不分大小写的。事实上，所有的符号名称不管原来的形式为何，都将被译成大写字母。但是盛群的 C 语言则有大小写之分。为了区分这两种语言的不同，凡是定义在 C 语言文件内而且被汇编程序使用到的变量及函数，都必须以大写字母来命名。

当 C 编译器将程序译成为汇编语言时，会在全局变量与 C 语言函数的名称前加上前缀下划线(underscore)。至于局部变量，如果只是宣告但未使用的局部变量，C 编译器不会为其保留存储器空间。可查看 C 编译器所生成的汇编语言文件，以找出 C 局部变量在编译后的名称。

全局变量

在 C 文件中的全局变量，编译后不会改变其名称大小写，但会在前缀加上下划线。

范例

```
TimerCt
TMP
```

编译后

```
_TimerCt
_TMP
```

局部变量

在 C 函数中的局部变量如果没有被其它程序所使用，则不会被译成汇编语言。可查看汇编语言文件找出结果为何。

```
void main(){
    int i, j, k;          ; k 未被使用
    long m;
    char c;
    i = j = m = c = 2;
    #asm
    set CR3[1].2          ; set bit 10 of m, i.e. m|= 0x400
    #endasm
}
```

汇编语言文件中对应的部分如下：

```
#line 2 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR1 DB ? ; i
#pragma debug variable 2 CR1 i
#line 2 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR2 DB ? ; j
#pragma debug variable 2 CR2 j
#line 3 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR3 DB 2 DUP (?) ; m
#pragma debug variable 2 CR3 m
#line 4 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR4 DB ? ; c
#pragma debug variable 2 CR4 c
```

第二与第三行表示 *i* 被译成汇编语言中的 *CR1*，同理 *j* 被译成为 *CR2*，*m* 被译成为 *CR3* 且 *c* 被译成为 *CR4*，*k* 没有被使用所以没有被编译。

注意： 如果加入新的局部变量或调整局部变量的顺序，那么编译后的名称会有所改变。

针对上述的范例，如果是具有多个数据存储器区块(memory bank)的单片机，则指令

```
set CR3 [1].2
```

可能会执行正确也可能错误。如果 CR3 被配置在较高编号的存储器区块，则程序会执行错误，但是这种现象是不可能发生的。因为局部变量被译成汇编语言时，会使用 **LOCAL** 伪指令定义此变量，同时指示编译器将变量配置在数据存储器的区块 0，因此就像定义在单一存储器区块的变量一样，可以正确的执行。

函数

就像全局变量一样，在 C 文件中的函数名称在编译时并不会改变字形的大小字，只是会加上前缀下划线。

范例

```
GetKey
IsBusy
```

译成

```
_GetKey
_IsBusy
```

函数的参数

C 文件中函数参数的名称被译成函数名称加上参数出现的序号，序号是从 0 开始计算的。

范例

```
GetKey (int row, long col)
```

row 被译成为 GetKey0

col 被译成为 GetKey1

参数的传递

由于单片机的资源限制，盛群 C 编译器通过 RAM 存储器取代堆栈传递参数至函数，函数参数的命名是以函数名称附加由 0 开始到所有参数数目的数字。就像局部变量一样，函数参数也是配置在数据存储器的区块 0。

范例

```
void function (int a, int b)
```

参数 a 将被译成为 function0，参数 b 将被译为 function1。使用两种程序语言写程序时，即使函数参数的数据类型超过一个字节，在汇编语言中仍应该把它宣告为 BYTE 的数据类型，例如参数为 WORD（2 个字节）的数据类型时，应该使用“DB 2 DUP(?)”来宣告。

返回值

C 函数的返回值会被存于 **A** 寄存器或 **RH** 系统变量。如果返回值的大小是一个字节（例如 **char**、**unsigned char**、**int**、**unsigned int**、**short**、**unsigned short**），则返回值被储存于 **A** 寄存器，如果返回值的大小是二个字节（例如 **long**、**unsigned long**、**pointer**）则高字节被储存于 **RH** 而低字节被储存于 **A** 寄存器。

注意： **RH** 变量位于 RAM 存储器的区块 0。

寄存器内容的保存

除了 ISR 函数，在所有使用汇编语言设计的函数中，不需要保存寄存器的内容。如果要用汇编语言设计 ISR 函数，则使用者有责任将 ISR 中所用到的寄存器的内容保存起来，当 ISR 函数执行完毕回到被中断之处前，将原先的数据回存给这些寄存器。

在 C 程序中调用汇编语言函数

此节将描述在 C 程序中调用汇编语言函数的步骤。此步骤共分为两个部分，一部分为汇编语言程序文件，另一部分为 C 语言程序文件。

→ **在汇编语言文件中：**

- 如果返回值为二个字节，则将 **RH** 宣告为外部字节变量。
- 将以下划线为前缀的函数名称宣告为公用函数(public)。
- 如果函数有参数，将这些参数宣告在数据存储器的区块 0，以及宣告为公用的参数。注意参数的命名。
- 将返回值放入 **A** 寄存器或 **RH** 系统变量。

→ **在 C 语言文件中：**

- 将以大写字母命名的函数宣告为外部函数(external)。
- 调用此函数。

下面的函数是定义在汇编语言文件中，被 C 程序所调用。

long KEYIN(int row, long col);

汇编语言文件：

```
;; 将变量 RH 宣告为外部变量
EXTERN RH:BYTE

;; 将函数与参数宣告为公用的
PUBLIC _KEYIN, KEYIN0, KEYIN1

;; 将函数参数安排在存储器的区块 0
RAMBANK0 KEYINDATA          ; 假设单片机具有多个 ram 区块(bank)
KEYINDATA .section 'data'
KEYIN0 DB?                  ;row
KEYIN1 DB 2 DUP (?)         ;col, 不要使用 'KEYIN1 DW ?'

;function body
CODE .section 'code'
_KEYIN:
...
MOV A, KEYIN0               ;取得参数 row 值
...
MOV A, KEYIN1               ;取得参数 col 的低字节
...
MOV A, KEYIN1[ 1 ]          ;取得参数 col 的高字节
...
;;将返回值放入寄存器 A 及系统变量 RH 中
MOV A, 0A0H                 ; 假设返回值是 0×A010
MOV RH, A                   ; 将高字节 0×A0 存入变量 RH
MOV A, 10H                  ; 将低字节 0×10 存入寄存器 A
RET
```

在 C 语言文件中：

```
// 将被调用的函数宣告为外部函数,同时以大写字母对此函数命名
extern long KEYIN(int row, long col);
long rc;
...
// 调用外部函数
rc = KEYIN(10, 20L);
```

在汇编程序中调用 C 函数

此节将描述在汇编程序中调用 C 函数的步骤。针对具有多个 ROM 存储器区块的单片机，在调用函数之前，必须要设定 BP 寄存器（存储器区块指针）。

→ **在 C 语言文件中**

- 宣告以大写字母命名的函数。

→ **在汇编语言文件中：**

- 如果返回值是二个字节，则宣告 **RH** 为外部的字节变量。
- 宣告以下划线为前缀命名的函数为外部函数。
- 如果函数需要参数，则将这些参数宣告为外部参数，要注意参数的命名规则。
- 如果函数有参数，将输入值设定给参数。
- 调用 C 函数。如果单片机只具备一个 RAM/ROM 存储器区块，则直接调用 C 函数。如果单片机具备多个 RAM/ROM 存储器区块，则先将 BP 寄存器设为函数所在的存储器区块，再调用 C 函数。

- 从 **A** 寄存器或系统变量 **RH** 中取出返回值。

下面的函数是以 C 语言撰写而被汇编程序所调用。

long KEYIN(int row, int col);

当单片机只具备单一的 ROM 存储器区块。

```
//-----------------------------------------------------
// In C file, 定义函数
//-----------------------------------------------------
long KEYIN(int row, long col){
...
}
void main( ){
...
}

;;-----------------------------------------------------
;; In assembly file
;;-----------------------------------------------------

;; 宣告 RH 为外部变量
EXTERN RH:BYTE

;; 宣告以下划线为前缀的函数名为外部函数
extern _KEYIN:near   ;;underscore and function name

;; 将函数的参数宣告为外部变量
extern _KEYIN0:byte  ; 参数 row
extern _KEYIN1:byte  ; 参数 col, 虽然其数据类型为 2 个 bytes,
                     ; 还是将其宣告为 BYTE

code_ki .section 'code'

;; 设定参数的输入值,准备调用函数 KEYIN(0x10, 0x200L)
mov a, 10H
mov KEYIN0, a        ; 将参数 row 设定为 10H
mov a, 2H
mov KEYIN1[1], a     ; 设定参数 col 高字节为 02H,低字节为 00H
clr KEYIN1           ;

;; 调用 C 的函数
call _KEYIN

;; 从寄存器 A 或系统变量 RH 中取得返回值
;; 寄存器 A 存放返回值的低字节,系统变量 RH 存放返回值的高字节
```

下面的例子中，函数定义在 C 语言文件中而被汇编程序所调用。

long KEYIN(int row, int col);

此单片机具有多个 ROM 存储器区块。

```
//-----------------------------------------------------
// In C file, 定义函数
//-----------------------------------------------------
long KEYIN(int row, long col){
...
}

;-----------------------------------------------------
; In assembly file
;-----------------------------------------------------

; 宣告 RH 为外部变量
     EXTERN RH:BYTE

     ; 宣告以下划线为前缀的函数名为外部函数
     extern_KEYIN:near

;; 将函数的参数宣告为外部变量
     extern_KEYIN0:byte  ; 参数 row
     extern_KEYIN1:byte  ; 参数 col, 虽然它是 2 bytes,
                         ; 但是仍要宣告为 BYTE

     code_ki  .section  'code'

     ;; 设定参数的输入值, 准备调用函数 KEYIN(0x10, 0x200L)
     mov a, 10
     mov KEYIN0,a        ; 参数 row
     mov a, 2
     mov KEYIN1[ 1 ],a   ; 参数 col 的高字节
     clr KEYIN1          ; 参数 col 的低字节

     ;;调用位于多个 ROM 存储器区块的函数
     ;;先要将 BP 寄存器设定为函数所在的存储器区块编号
     mov a, bank _KEYIN
     mov bp, a           ; 更改区块编号 bank number
     call _KEYIN         ; 调用函数

     ;; 从寄存器 A 或系统变量 RH 中取得返回值
     ;; 寄存器 A 存放返回值的低字节,系统变量 RH 存放返回值的高字节
```

使用汇编语言撰写 ISR 函数

ISR（中断服务程序）是由硬件中断所调用，不可由程序自行调用，因此不会有参数的输入，也不会有值返回。如果用汇编语言撰写 ISR 函数，则不会与 C 语言的程序有关系，只需要将此汇编语言文件加入项目即可。若需要更多关于撰写 ISR 程序的资料，请参考汇编语言的使用说明。

不论 ISR 是由汇编语言或 C 语言所撰写，都不要从 ISR 中调用 C 函数。

第三章

用 C 语言编程

3

此章节包括以下部分：

- 开始一个 C 程序
- 定义中断服务入口向量(ISR)
- 在程序存储器中定义表格与标号
- 在数据存储器内定义变量
- 单片机特殊功能寄存器
- 内置功能
- 编程小窍门
- 串口通信例程
- 程序主要架构
- 数据类型

开始一个 C 程序

在 HT-IDE3000 的项目中编写源程序可以使用汇编语言或者 C 语言。芯片复位后，程序总是从程序存储器的地址 0 开始执行。当项目中有 C 源程序并且程序入口在 C 程序中，那么 C 编译器会将函数 *main* 强制放在程序空间的地址 0 处。并且，需要遵循以下原则，

- 一旦在一个源程序文件中定义了函数 *main* 那么在其他的源程序文件中不能再定义同一名称的函数 *main*。函数 *main* 是程序执行的入口。
- 程序存储器的地址 0 不能为其他功能使用，诸如查表或者代码段。这个地址只能被函数 *main* 所使用。建立一个项目时，以下的语句会产生一个错误：
- **#pragma vector** ResetFunction @ 0x00

函数 ResetFucntion 不能够定义在地址 0

范例：

```
void test(){
}
void main(){          // define the main function
     test();
}
```

定义中断入口向量

当项目中要使用单片机中断以及相关的中断服务程序(ISR)，那么在编写相关的 C 程序中要注意适当的使用以及限制。

没有必要保存系统寄存器，Holtek 的 C 编译器会自动地将使用到的系统寄存器进行保存。

在 C 源程序文件中宣告中断服务子程序的名称和地址

宣告子程序的名称和地址可以像如下使用语句 **pragma vector**

#pragma vector *IsrRoutineName* @ *address*

pragma 和 **vector** 是关键字。
IsrRoutineName 是中断服务子程序的名字。
Address 是中断服务子程序在程序存储器中的入口地址。地址 0 是为函数 *main* 保留的不能使用。

在 C 源程序文件中定义 ISR

准备定义的函数名要与之前用**#pragma vector** 定义的 *IsrRoutineName* 的名相同。

```
#pragma vector _ExternISR @ 0x04

void _ExternISR(void){

}
```

限制

使用 C 语言编写 ISR 时，要记住有以下几项限制

- ISR 没有参数并且返回类型为 **void**。
- ISR 不可以重复进入，不要在 ISR 内允许中断。
- 在程序中不需要自己去调用。当中断发生的时候，系统会自己响应并执行。
- 不要在 ISR 内调用任何 C 程序中用户定义的函数，但是调用内置的函数没有关系。如果 ISR 希望调用一个函数，那么这个函数必须是使用汇编语言编写的。
- 如果 ISR 内包含嵌入汇编指令，那么由于执行这些指令而影响到的寄存器就需要在执行这些指令之前预先保留，待执行完毕恢复寄存器。因为 Holtek 的 C 编译器只保存由于 C 语句造成影响的寄存器。

范例：

```
#include <ht47c20.h>
#pragma      vector _ExternISR     @ 0x4
```

```
#pragma      vector _TimeBaseISR   @ 0x8
#pragma      vector _RTCISR        @ 0xc
#pragma      vector _TimerISR     @ 0x10

unsigned count;

void _ExternISR(void){
}

void _TimeBaseISR(void){
  count=count>>7|count<<1;
  _pa = count;
}

void _RTCISR(void){
}

void _TimerISR(void){
}

void main()
{
  count=0xee;
  _intc0=0x05;      //EMI&ETBI ENABLE
    while(1);
}
```

在程序存储器中定义表格与标号

Holtek 的 C 编译器给带固定值的表格以及标号分配程序存储器空间。当一个表格（数组）或者标号带固定值，那么可以通过宣告其类型为 **const** 来给它在程序存储器分配空间。这些标号以及数组的用法与在数据存储器中定义的数组或标号的用法一样，它们的区别就是在程序存储器内定义的标号或数组是不能修改的，这些常数的最大值为 255，如果需要更大的数值，它们可以分为好几个常数。

范例：

```
// below three variables are in Program Memory
const unsigned char ascii[16]="0123456789ABCDEF";
const unsigned char pattern[16]={0,1,2,3,4,5,6,7,8,
                                 9,10,11,12,13,14,15};
const unsigned int cl = 0x8B;

// below variables are in Data Memory
#pragma rambank0
unsigned char str[2];

void itoa(unsigned int v, unsigned char *s){
  *s = ascii[v & 0xf];
  _swap(&v); //swap nibble
  *(s+1) = ascii[v & 0xf];
}

void main(){
  unsigned int val;
```

```
  val = cl;
  itoa(val, str);
}
```

注意：在宣告时，const 类型的标号或者数组需要初始化，一个 const 类型的数组长度也需要注意。

在数据存储器内定义变量

在 Holtek 单片机的数据存储器中有两个或者三个功能块：特殊存储器空间，通用数据存储器空间以及 LCD 数据存储器空间。数据存储器可能包含不止一个区块，数据存储器区块 0（bank0）以及其他的数据区块空间。

特殊寄存器空间是常驻单片机存储器的功能寄存器。LCD 数据存储器空间保存的数据是为 LCD 显示用的。在执行程序时，通用数据存储器空间为变量提供空间。所有的 Holtek 单片机都拥有包括特殊功能寄存器和数据存储器区块 0 的空间。只有某些单片机包含不止一个的数据存储器区块。详细情况，请参考相应的资料。

如果没有特别明确的指出地址，那么一个变量的地址由 C 编译器在通用数据存储器中分配，并且这个地址是可以随机改变的。对于特殊存储器和 LCD 数据存储空间来说，需要给出明确的专有地址，否则它将被认为是一个随机变量。

指定变量地址

在数据存储器中，可使用操作数'@'指定变量地址。

- 语法

  data_type varaible_name @ memory_address

- 描述

memory_address 指定变量 *variable_name* 的地址，它包含数据存储器的区块地址以及在该区块中的确切地址。*memory_address* 的高位表示数据存储器的区块数值，低位表示在该区块内的地址。*data_type* 则指定数据类型。

范例

```
int v1 @ 0x50;  // v1 is in address 0x50 of RAM bank 0
int v2 @ 0x380; // v2 is in address 0x80 of RAM bank 3
int v3 @ 0xef0; // v3 is in address 0xf0 of RAM bank 14
```

在多个数据存储器区块访问变量

在汇编语言中，必须通过设置区块指针并且间接寻址来访问高位数据存储器区块中的变量。Holtek 的 C 编译器已经自动的做好了这些工作，用户不需要再费力地去做这件事情了。

- 范例

```
int v1 @ 0x5B;   // v1 is in address 0x5B of RAM bank 0
int v2 @ 0x2F0;  // v2 is in address 0xF0 of RAM bank 2
int v3;          // bank number is unknown

void main(){
    v1 = 10; //access bank 0 variable
    v2 = 10; //access high bank variable
    v3 = 10;
}
```

在程序存储器区块 0 中指定变量(提高性能)

对于具有多个区块的单片机来说，它必定需要通过间接寻址来完成对其他高位区块数据存储器变量的访问。越多的访问，则会需要执行越多的指令，造成的结果就是会大大地降低程序的效率。因此，最好将访问频率较高的变量定义在数据存储器的区块 0 内。通过使用 C 伪指令 **pragma rambank0**，可以将访问频率较高的变量定义在数据存储器的区块 0。它会强制 Holtek 的连接器为指定的变量在数据存储器区块 0 中找存储空间，如果数据存储器区块 0 没有足够的空间而其他区块有足够的空间可以放置这些变量的话，连接器将会发出一个连接错误。在这种情况下，用户需要重新安排数据存储器区块 0 的变量。伪指令 **pragma norambank** 将终结 **rambank0** 功能。除非数据存储器区块 0 耗尽了，所有需要在 ***rambank0*** 和 **norambank** 宣告的变量都会放置在数据存储器的区块 0 中。

- 语法

#pragma rambank0
//数据宣告 ： 在此区块中定义的变量会被放置在数据存储器区块0

#pragma norambank
//数据宣告 ： 在此定义的变量没有必要一定放置在数据存储器区块0

- 描述

关键字 **rambank0** 会直接告知编译器在它之后宣告的变量需要放置在数据存储器区块 0 中，直到关键字 **norambank** 或者此源程序文件结束。如果该单片机只有一个数据存储器区块的话，那么编译器会忽略这两个关键字。

- 范例

```
// default is norambank
unsigned int v1;          //v1's bank number is unknown

//switch to rambank0
#pragma rambank0
unsigned int i, j;        //i, j located at RAM bank 0
long len;                 //len located at RAM bank 0

//In rambank0 area the address cannot be larger than 0x100
unsigned char uc0 @ 0x83;
```

```
// back to norambank
#pragma norambank
unsigned int iflag;     //bank number of iflag is unknown
unsigned char uc @ 0x140;

//switch to rambank0 linking mode
#pragma rambank0
bit bitflag;          //bit variable should always be
                    // declared in rambank0 block

void main(){
    i = 1;          //MOV A,1
                    //MOV _i, A

    iflag = 1;      //MOV A,BANK _iflag
                    //MOV [04H],A
                    //MOV A,OFFSET _iflag
                    //MOV [03H],A
                    //MOV A,1
                    //MOV [02H],A
    uc0 = uc = 0;
    bitflag = 1;
}
```

指针范围

在做指针运算时，指针不能访问整个区块范围。当指针有溢出时，它将重叠。

- 范例

```
#pragma rambank0
unsigned char *p1;
unsigned long *p2;

void main(){
    p1 = (unsigned char *)0x2f0;
    p1 += 0x20; //now p1 points to address 0x210, not 0x310

    p1 = (unsigned char *)0x100;
    p1--;  //now p1 points to address 0x1ff, not 0xff

    p2 = (unsigned long*)0x3fe;
    p2++; // 'long' occupies two bytes.
          // now p2 is pointed to 0x300 not 0x400.
}
```

因为指针变量不能访问整个区块范围，Holtek 的 C 不支持长整型指针变量的运算。

.

- 范例

```
#pragma rambank0
unsigned char *p1, *p2;
unsigned int i;
unsigned long len;
```

```
void main(){
    p1 = p2+10; //ok
    p1 = p2+0x100; //error, 0x100 is a long integer
    p1 += i; //ok
    p1 += len; //error, len is a long integer
}
```

访问 LCD 数据存储空间

Holtek C 提供一个非常方便的访问 LCD 数据存储空间的方式。将相应的 LCD 数据存储器变量在 Specify **Address To a Variable** 中使用操作数'@' 定义。以下的例子将给出宣告和如何访问 LCD 数据存储器。

.

- 范例 :

```
// LCD data memory is at RAM bank 14 (0x0e)
// lcd_day is at address 0x80 of RAM bank 14
// lcd_mon is at address 0x82 of RAM bank 14

#include <HTG2190.H>
// delcared lcd_day at LCD data area
unsigned char lcd_day @ 0xe80;

//declared lcd_mon at LCD data area
unsigned char lcd_mon @ 0xe82;

#pragma rambank0
unsigned int i, j;
unsigned char *lcd_ptr;

/*
Delcared non RAM bank 0 variables
A realistic scenario is that the variables are declared within the rambank0 block
if the memory is available.
*/

#pragma nonrambank0
unsigned int tmp;

void main(){
   lcd_mon = 0x10;              // put value 0x10 to LCD data memory 0xe82
   lcd_ptr = &lcd_day;          // lcd_ptr points to LCD data memory 0xe80
   * lcd_ptr = 0xff;            // put value 0xff to LCD data memory 0xe80
   *(lcd_ptr+1) = 0xa0;         // put value 0xa0 to LCD data memory 0xe81
}
```

单片机特殊功能寄存器

Holtek 的单片机特殊功能寄存器在数据存储器区块 0 的前面部分。这个数据存储器不能作为通用变量使用。

访问特殊功能寄存器

要访问特殊功能寄存器，必须用一个变量来限定这个寄存器。Holtek C 提供一

个非常简便的方式来访问所有特殊功能寄存器的字节或者位。

- 字节变量

定义一个字节的特殊功能寄存器变量的语法与定义一个数据变量确定位置的方法一样。

data_type varaible_name @ memory_location

推荐宣告数据类型为"unsigned char"。例如，

```
unsigned char _a      @ 0x05;
unsigned char _pcl    @ 0x06;
unsigned char _tblp   @ 0x07;
unsigned char _tblh   @ 0x08;
unsigned char _wdts   @ 0x09;
unsigned char _status @ 0x0a;
unsigned char _intc   @ 0x0b;
unsigned char _tmr0h  @ 0x0c;
unsigned char _pa     @ 0x12;
unsigned char _pb     @ 0x14;
```

特殊功能寄存器的使用方法与普通的数据变量的方法一样，例如：

```
_pa = 0xff; //set PA
if (_pb == (unsigned char)0x80){
...
}
```

- 位变量

Holtek 的 C 编译器为特殊功能寄存器提供内置的位变量。这些位变量的命名规则如下：

_xx_n

xx: 特殊功能寄存器的地址，两位十六进制数

n: 特殊功能寄存器的位数值

例

_0a_0是地址0aH的第0位的位变量，状态寄存器的进位位
_12_1是地址12H的第1位的位变量，端口A

在使用内置的位变量之前没有必要宣告它们，例如，利用*#define* 用户可以给这些位变量分配一个有意义的名称，如下所示：

// The HT48C50-1

```
#define _c          _0a_0
#define _ac         _0a_1
#define _emi        _0b_0
#define _eei        _0b_1
#define _et0i       _0b_2
#define _et1i       _0b_3
#define _eif        _0b_4
#define _t0f        _0b_5
#define _t1f        _0b_6
```

```
#define _pa0        _12_0
#define _pa1        _12_1
#define _pa2        _12_2
```

这些变量的数据类型是位变量。它的使用方法与普通的位数据变量一样，例：

```
bit bflag;
...
_emi = 1;   //enable interrupt
_c = 1;     //set carry
if (_pa0){ //if port A bit 0 set
...
}
bflag = _eei ;
_pa0 = _pa2; //bit assignment
_pa1 = bflag;
```

对于每一个 Holtek 单片机来说，都有相对应的包含文件(include file)来宣告这些单片机的特殊功能寄存器。这些包含文件的文件名与单片机的名称一样。例如， HT48C10-1.H 就是单片机 HT48C10-1 的包含文件。要访问一个特殊功能寄存器，可以包含一个正确的单片机包含文件或者单独宣告特殊功能寄存器。

下面给出如何访问特殊功能寄存器的例子，单片机类型是 HT48C10-1。

```
#include <HT48C10-1.H>

void main(){
    int i;
  _intc = 0;
  _tmrc = 0;
    _tmr = 0;
  _c = 0;       //clear carry flag
  _rrc(&i);    //rotate right through carry
  ...
}
```

访问输入输出端口

用户可以用访问特殊功能寄存器的方法来访问输入输出端口。它包含字节变量和位变量。

例：

```
unsigned char _pac @ 0x13;
unsigned char _pbc @ 0x15;
#define _pa0 _12_0
#define _pa3 _12_3
#define _pa5 _12_5
#define _pb3 _14_3
#define _pc2 _16_2
#define _pc5 _16_5

void main(){
  _pac = 0xff;     // set port A control register
  _pbc = 0x40;     // set port B control register
```

```
  _pa0 = 1;        // set port A bit 0
  _pb3 = 0;        //clear port B bit 3
  _pc5 = _pa3;
  if (_pa5){       //if bit 5 of port A == 1
  ...
  }
  while(! _pc2){   //while bit 2 of port C == 0
  ...
  }
}
```

内置函数

Holtek 的 C 编译器提供一些内置的函数，与直接用汇编指令编写的相似。一些内置函数编译后只用一条汇编指令，而另一些内置函数则可以更方便地使用 C 语言编程。

类似汇编语句的内置函数

以下的内置函数，Holtek 的 C 编译器会将其编译为相应的汇编指令。

C 子程序	汇编指令
void _clrwdt( )	CLR WDT
void _clrwdt1( )	CLR WDT1
void _clrwdt2( )	CLR WDT2
void _halt( )	HALT
void _nop( )	NOP

- 例：

```
//assume the watchdog timer is enabled
//and use one clear WDT instruction

void dotest(){
...
}

void main(){
       unsigned int i;
       for(i=0; i<100; i++){
             _clrwdt();          // CLR WDT
             dotest();
       }
}
```

- 例：

```
//assume the watchdog timer is enabled
//and use two clear WDT instructions
```

```
void dotest(){
...
}

void main(){
       unsigned int i;
       for(i=0; i<100; i++){
             _clrwdt1();       // CLR WDT1
             _clrwdt2();       // CLR WDT2
             dotest();
       }
}
```

移位函数

在 C 语言中没有移位操作数，然而 Holtek 的 C 编译器提供了内置的数据移位函数。

```
void _rr(int*);     //rotate 8 bits data right
void _rrc(int*);    //rotate 8 bits data right through carry
void _lrr(long*);   //rotate 16 bits data right
void _lrrc(long*);  //rotate 16 bits data right through carry
void _rl(int*);     //rotate 8 bits data left
void _rlc(int*);    //rotate 8 bits data left through carry
void _lrl(long*);   //rotate 16 bits data left
void _lrlc(long*);  //rotate 16 bits data left through carry
```

例，

```
#include <HT48C50-1.h>
unsigned int ui;
unsigned long ul;

void error(){
  while(1);
}

void main(){
  ui = 0x1;
  _rr(&ui);    //rotate right
  if (ui != (unsigned int)0x80) error();
  _c = 1;      //set carry
  _rrc(&ui);   //rotate right through carry
  if (ui != (unsigned int)0xc0) error();
  ul = 0xc461;
  _lrl(&ul);   //long rotate left
  if (ul != 0x88c3) error();
  _c = 0;      //clear carry
  _lrlc(&ul); //long rotate left through carry
  if (ul != 0x1186) error();
}
```

高低位交换函数

```
void _swap(int *); //swap nibbles of 8 bit data
```

范例,

```
unsigned int ui;

void error(){
```

```
  while(1);
}

void main(){
  ui = 0xab;
  _swap(&ui);
  if (ui != (unsigned int)0xba) error();
}
```

延迟周期函数

```
void _delay(unsigned long)
```

_delay 函数会使单片机执行指定的指令周期数值。值为 0 时会引起一个死循环。_delay 的参数只能为一个常数数值，它不接受一个变量

例如，

```
#define _pa0 _12_0 //port A bit 0
unsigned char _pb @ 0x14 //port B

void error(){
  _delay(0);   //infinite loop. same as while(1);
}

void main(){
  unsigned long time;
//wrong, parameter should be constant value only



  _delay(0);   //infinite loop. same as while(1);
}

void main(){
  unsigned long time;
//wrong, parameter should be constant value only
  //_delay(time);

  _pa0 = 1;
  _delay(1); //delay 1 instruction cycle
  _pa0 = 0;
  _delay(15); //delay 15 instruction cycle
  if (_pb != (unsigned int)0x8f) error();
}
```

编程提示

定义变量为无符号数据类型

通常，无符号变量的运算要比有符号变量的运算简单。所以如果一个变量没有负值时，建议将其定义为无符号数据类型。

范例

```
int i,j;
unsigned int ui, uj;

void test(){
       if (i >= j);       // translate to 8 instructions

       if (ui >= uj);     // translate to 4 instructions
}
```

第一个有符号数的比较被翻译成 8 个汇编指令，而第二个被翻译成 4 条指令。

将变量定义在数据存储区块 0

位于数据存储区块 0 以上区块的数据需要间接寻址访问，因此会产生一些低效率的代码。所以对于有着多个数据存储区块的单片机，最好将使用频繁的变量定义在数据存储区块 0 里.

范例

```
//file RAMBANK0.C
//assume the MCU has multiple RAM banks

#pragma rambank0
unsigned int ui0;  // ui0 is in RAM bank 0

#pragma norambank
unsigned int ui;   // ui is relocatable, may not in RAM
                 // bank 0

void test(){
       ui0++;      // translate to 1 instruction
       ui++;       // translate to 5 instructions
}
```

当在另一个源文件的程序需要访问到定义在数据存储区块 0 的变量，需要谨慎使用。如果一个变量在文件 RAMBANK0.C 中被定义在 RAM bank 0 里，它可以在其他文件中的程序被访问，如 ACCESS0.C and ACCESS1.C 。但是这个变量需要被声明为外部变量并且位于数据存储区块 0，否则会产生多余的和不正常的代码。而执行结果是不可预料的。

范例

```
// assume the ui0, ui are declared in the above example
// file RAMBANK0.C

// file ACCESS0.C
// declare variables to be the same as RAMBANK0.C

#pragma rambank0
extern unsigned int ui0;   // declare ui0 in RAM bank 0

#pragma norambank
extern unsigned int ui;

void testB(){
       ui0++;   // translate to 1 instruction;  correct
       ui++;    // translate to 5 instructions; correct
```

```
}

// file ACCESS1.C
// declare variables to be not the same as RAMBANK0.C

#pragma rambank0
extern unsigned int ui;  // declared ui in rambank0

#pragma norambank
extern unsigned int ui0;

void testC(){
       ui0++;       // 5 instructions; correct
       ui++;        // 1 instruction;  wrong
}
```

在 ACCESS1.C 文件中，语句 ui0++会被翻译成 5 条指令，比 ACCESS0.C 文件中多出了 4 条。不过这些语句执行的结果是正确的。但是语句 ui++被翻译成一条指令且执行结果是不可预料的。原因是 ui 在文件 RAMBANK0.C 中没有定义在 RAM bank 0 里，应该通过间接寻址来访问。

定义位变量

位变量占用一位的存储单元。如果一个变量只有两个值，那么定义成位变量是很合适的。这不仅使用了较小类型的数据，也有生成代码少的优点

范例

```
//assume the MCU has single RAM bank
bit bitflag;
unsigned int intflag;

void test(){
       bitf = 1;          // 1 instruction
       intflag = 1;       // 2 instructions

       if (bitflag);      // 2 instructions
       if (intflag);      // 3 instructions
}
```

分配地址给指针

如果要分配常量地址给指针，类型需要明确指定，否则编译器将认为出错。

范例

```
//assume the MCU has multiple RAM banks
int *p1;
unsigned char *p2;
long *p3;

void main(){
       //point to RAM bank 0, offset 0x50
       p1 = (int*)0x50;
       p1 = 0x50;     // error, no casting
```

```
        //point to RAM bank 1, offset 0x60
        p2 = (unsigned char *)0x160;

        //point to RAM bank 2, offset 0x30
        p3 = (long *)0x230;
}
```

使用更有效的方法获得模数

当你想得到一个除法的商和余数时，下面的语句是最常用的。

```
q = d1 / d2;
r = d1 % d2;
```

每个语句都会调用除法子程序，总共两次。其他的获得商和余数的方法是使用内嵌的汇编程序。求商的语句相同，但是求余数语句换作内嵌的汇编程序。对于 8 位有符号/无符号除法，余数会被存储在系统变量 T3 中，对于 16 位有符号/无符号除法，余数会被存储在系统变量 T4 和 T5。T4 是高字节，T5 是低字节。

→ 只有一个数据存储区块的单片机

- 8 位除法

```
unsigned int d1, d2;
unsigned int q, r;
q = d1 / d2;            // get quotient
#asm
MOV A, T3               ; get remainder
MOV _r,A
#endasm
```

- 16 位除法

```
unsigned long d1, d2;
unsigned long q, r;
q = d1 / d2;
#asm
MOV A, T5
MOV _r,A          ; get low byte remainder
MOV A, T4
MOV _r[1], A      ; get high byte remainder
#endasm
```

→ 有多个数据存储区块的单片机.

- 8 位除法, r 在数据存储区块 0

```
unsigned int d1, d2;
unsigned int q;
#pragma rambank 0
unsigned int r;
#pragma norambank

q = d1 / d2;
#asm
MOV A, T3
MOV _r,A
#endasm
```

- 16 位除法, r 在数据存储区块 0

```
unsigned long d1, d2;
unsigned long q, r;
q = d1 / d2;
#asm
MOV A, T5
MOV _r,A      ;get low byte remainder
MOV A, T4
MOV _r[1], A  ;get high byte remainder
#endasm
```

- 8 位除法, r 不在数据存储区块 0

```
unsigned int d1, d2;
unsigned int q;
unsigned int r;

q = d1 / d2;
#asm
MOV A, 0E0H
AND [04H],A    ;BP, clear RAM bank and preserve ROM bank
MOV A, BANK _r
OR  [04H], A   ; set bank pointer
MOV A, OFFSET _r
MOV [03H], A   ; move offset to MP1
MOV A, T3
MOV [02H], A   ; move T3 to R1
#endasm
```

- 16 位除法, r 不在数据存储区块 0

```
unsigned long d1, d2;
unsigned long q, r;
q = d1 / d2;
#asm
MOV A, 0E0H
AND [04H],A       ;BP, clear RAM bank and preserve ROM
bank
MOV A, BANK _r
OR  [04H], A      ; set bank pointer
MOV A, OFFSET _r
MOV [03H], A      ; move offset to MP1
MOV A, T5
MOV [02H], A      ; store to low byte of remainder
INC [03H]         ; point to high byte of remainder
MOV A, T4
MOV [02H], A      ; store to high byte of remainder
#endasm
```

常量变换 / 强制转换

Holtek C 编译器是 8 位编译器. 注意 int 是等同于 char 数据类型的，数值范围是ñ128 至+127. 如果应用要处理 8 位常量整型数据，则需要要将它强制转换成 int/char(或者 unsigned int / unsigned char),否则一个 8 位的十六进制的整型数据可能会被错误的转换成 16 位的整型数据。如果没有外部类型强制转换，介于 0x80 和 0xff 之间的常量会被转换成相应的 16 位整型数据而不会符号扩展。

范例:

- 有外部类型转换, (unsigned int)0xff 相当于一个无符号 8 位整型数据，数值是 255
- 有外部类型转换, (int)0xff 相当于一个有符号 8 位整型数据，数值是 ñ1
- 没有外部类型转换, 0xff 会被暗中地转化成一个 16 位长整型数据，数值是 255

范例

```
//assume the MCU has a single RAM/ROM bank
unsigned int ui;
int i;

void main(){
       //8 bit signed comparison
       //5 instructions
       if (i >= 0x7f){
             //equals to if (i >= 127)
       }


       //0x80 implicitly converted to (long)128
       //16 bit signed comparison
       //16 instructions
       if (i >= 0x80){
             // equals to if (i >= 128)
             //always false
       }


       //explicitly casting 0x80 to (int)-128
       //8 bit signed comparison
       //5 instructions
       if (i >= (int)0x80){
             // equals to if (i >= -128)
             //always true
       }


       //8 bit unsigned comparison
       //4 instructions
       if (ui >= 0x7f){
              // equals to if (ui >= 127);
       }

       //0x80 implicitly converted to (long)128
       //16 bit signed comparison
       //14 instructions
       if (ui >= 0x80){
              //equals to if (ui >= 128L)
       }

       //explicitly casting 0x80 to (unsigned int)128
       //8 bit unsigned comparison
       //4 instructions
       if (ui >= (unsigned int)0x80){
       }
}
```

串行端口传输范例

这个范例将告诉你如何使用 Holtek C 语言编写时间要求严格的程序。因为翻译 C 语言生成怎样的指令代码取决于编译器，下面范例中的延时常数可能会因为不同版本的编译器而不同。所以在第一次使用之前必须检查延时常数，相应的更新 C 编译器和单片机型号。由 C 编译器产生的指令取决于程序存储器/数据存储器是单区块的还是多区块的。

初始程序

串行端口传输协议规定一个启始位 0、8 位数据位、一位结束位 1。下面是针对单区块数据存储器单片机的初始程序。

```
// set address 0x12 bit 1 to be output pin (PA1)
#define tx  _12_1

unsigned char sent_val;

void main(){
    _13_1 = 0; //set PA1 as output pin
    sent_val = 'a';
    transmit();
}

void transmit(){
  unsigned char sent_bit;
  unsigned char i;

  tx = 0;             // L1 start bit
  for(i=0; i<8; i++){
      sent_bit = sent_val & 0x1;
      sent_val >>= 1;

      if (sent_bit){
          tx = 1;      // L2
      }
      else {
          tx = 0;      // L3
      }
  }
  tx = 1;             // L4 stop bit
}
```

上例中的函数 transmit( )在传送的波特率上是不正确的。为了匹配传输波特率，需要计算合适的延迟时间插入传送每一位之前或之后。因为汇编语言的输出 0 和 1 的语句是不同的，所以最好使用不同的 C 语句分别输出 0 和 1。因此建议将语句

```
tx = sent_bit;
```

替换成

```
if (sent_bit){
    tx = 1;      // L2
}
else {
    tx = 0;      // L3
```

```
}
```

语句 tx = sent_bit 不能决定是传送 0 还是传送 1。

调节传输时序

现在我们需要调节时序以使得所有传输过程(L1 到 L2、 L1 到 L3、 L2 到 L3、L3 到 L2、 L2 到 L4 或 L3 到 L4)的指令周期数都相同。在 HT-IDE3000 编译程序后，除错窗口将被激活。

→ **调节 L1 到 L2 和 L1 到 L3**

- 在 L1, L2, L3 处设置断点,打开 View 菜单下的 Cycle Count 窗口。
- 运行 ICE 后会停止在 L1 处。
- 在 Windows 菜单下的 Watch 窗口[1]中，更改 sent_val 数值为 1。
- 复位 Cycle Count 的值。
- 运行 ICE 后会停在 L2 处， cycle count = 0x11。
- 复位 ICE.
- 运行 ICE 后会停止在 L1 处。
- 在 Watch 窗口中，更改 sent_valís 数值为 0。
- 复位 Cycle Count 的值。
- 运行 ICE 后会停在 L3 处，cycle count = 0x12。

现在，我们知道 L1 和 L2 间的指令周期比 L1 和 L3 间的指令周期多 1 个。因此应该在 L2 之前增加一个指令周期的延迟，这样从 L1 到 L2 和从 L1 到 L3 的指令周期都等于 0x12。

```
if (sent_bit){
    _delay(1);      // add this statement
   tx = 1;         // L2
}
```

→ **调节 L2 到 L3 和 L3 到 L2**

使用已修改的代码来做下面的测试

- 在 L1、L2、L3 处设置断点。
- 运行 ICE 后会停止在 L1 处。
- 在 Watch 窗口中，更改 sent_val 数值为 5(00000101b)。
- 运行 ICE 后会停在 L2 处。
- 复位 Cycle Count 的值。
- 运行 ICE 后会停止在 L3 处，cycle count = 0x12。
- 复位 Cycle Count 的值。
- 运行 ICE 后会停在 L2 处，cycle count = 0x10。

现在，L2 和 L3 间的指令周期是 0x12, L3 和 L2 间的指令周期是 0x 10。因此需要在 L3 之后增加两个指令周期的延迟。

```
else {
   tx = 0;          // L3
     _delay(2);      // add this statement
```

[1] 在HT-IDE3000 Watch窗口中，输入点sent_val (.sent_val) 再按回车键。可以看到如 ì.sent_val :[xxH] = nnî这样的显示内容. 这时就可以将nn修改成01了。注意不要忘记在完成修改后按回车键否则数值不会被改变。.

```
    }
```

在 L3 之前增加延迟是错误的，因为这样会延长 L1 到 L3 的时间。

→ **调节 L2 到 L4 和 L3 到 L4**

至此，(L1,L2), (L1,L3), (L2,L3), (L3,L2)区间里的指令周期都已经相同。只剩下 L2 和 L4 需要检查。使用一更改的程序作下面的测试。

- 在 L1、L2、L4 处设置断点。
- 运行 ICE 使程序停至 L1。
- 在 Watch 窗口中，更改 sent_val 数值为 0x80。
- 运行 ICE 使程序停至 L2。 (最后的循环)
- 复位 Cycle Count 的值。
- 运行 ICE 使程序停至 L4。cycle count = 0x8。

L4 之前的延迟指令周期应该是 10 (0x12-0x8)。

```
    _delay(10);    // add this statement
    tx = 1;        // L4 stop bit
```

至此所有的传输时间都是相同的 18 个指令周期了。

波特率匹配调节

波特率 =系统时钟频率 / 4 / (传输一位所需指令周期数)
传输一位所需指令周期数= X+18, X 是额外的延迟指令周期。
所以 X 的求取公式为

$$X = (\text{系统时钟频率} / \text{波特率} / 4) - 18$$

例如, 系统时钟频率= 4MHz 且波特率= 9600 则 X 等于 86
下面是最终得到的程序。

```
// This function depends on compiler and MCU.
// You MUST adjust the delay constants when different
// compiler or MCU are used

// suppose address 0x12 bit 1 is the output pin (PA1)
#define tx   _12_1

unsigned char sent_val;

void transmit(){
  unsigned char sent_bit;
  unsigned char i;

  tx = 0; // L1 start bit
  for(i=0; i<8; i++){
     sent_bit = sent_val & 0x1;
     sent_val >>= 1;
       _delay(86);          // add this statement
     if (sent_bit){
          _delay(1);        // add this statement
        tx = 1;           // L2
     }
     else {
        tx = 0;           // L3
        _delay(2);        // add this statement
```

```
        }
    }
    _delay(86+10);            // add this statement
    tx = 1;                   // L4 stop bit
    _delay(86);               // add this statement
}
```

接收部分和上面类似。

框架程序范例

```
//include files
#include <ht49C50-1.h>

//Interrupt service routines declaration
#pragma vector external_isr @ 0x4
#pragma vector timer0_isr @ 0x8
#pragma vector timer1_isr @ 0xc

//RAM bank undefined variables
unsigned int uia, uib;
unsigned long ula, ulb;

//RAM bank 0 variables
#pragma rambank0
unsigned int uia0, uib0;
unsigned long ula0, ulb0;
bit flag;

//ISR
void external_isr(){
}

void timer0_isr(){
}

void timer1_isr(){
}

//main function
void main(){
}
```

数据类型

数据类型

下面的表格列出了数据类型、大小与范围：

数据类型	大小	范　围
bit	1	0, 1
char	8	-128~127
unsigned char	8	0~255
int	8	-128~127
unsigned	8	0~255
short int	8	-128~127
unsigned short int	8	0~255
long	16	-32768~32767
unsigned long	16	0~65535

浮点数据类型不支持。

第四章

C 程序范例

4

以下例子的源程序项目都可以在 HT-IDE3000 的安装目录的子目录<*Sample\C Example*>找到。

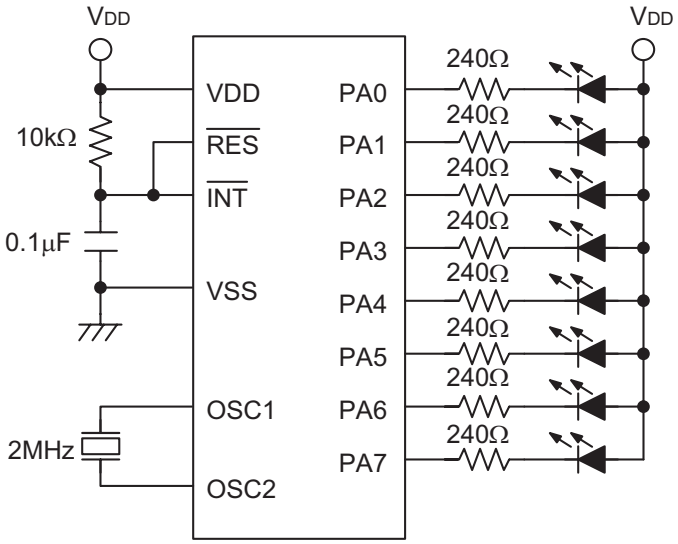
输入/输出应用

扫描灯

这个范例是对 LED 阵列进行扫描的功能仿真。在此例中，一排 LED 将被依次点亮。硬件电路使用了 PA 口的 PA0~PA7，每个引脚都通过一个 240 欧姆的电阻串联到一个 LED。

→ 电路设计

PA0~PA7 为输出引脚, 每个引脚都通过串联一个 240Ω的电阻控制一个 LED。通过右移和左移运算将点亮的 LED 由左至右和由右至左移动。具体电路参见电路图。



→ 程序

```
//Scan.c
//
//Body: HT48C10-1
//Mask option
//All the mask options use the default value.

#include <ht48c10-1.h>

bit direction;
unsigned char lamp;

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
     _intc = 0;
     _tmrc = 0;
     _tmr = 0;
     _pac = 0xff; //input mode
     _pbc = 0xff;
     _pcc = 0xff;
}
```

```
void main(){

    safeguard_init();

    direction = 0; //shift left direction
    _pac = 0;      //set port A as output port
    lamp = 1;      //set initial lamp light up

    while(1) {
       _pa = lamp;  //output lamp value to port A

       _delay(50000);

       if(!direction)
           lamp <<= 1;
       else
           lamp >>= 1;

       if(lamp & (unsigned char)0x80)
           direction = 1;  //shift right
       else if(lamp & 0x01)
           direction = 0;  //shift left
       //else, don't change the direction
    }
}
```

→ **掩膜选项**

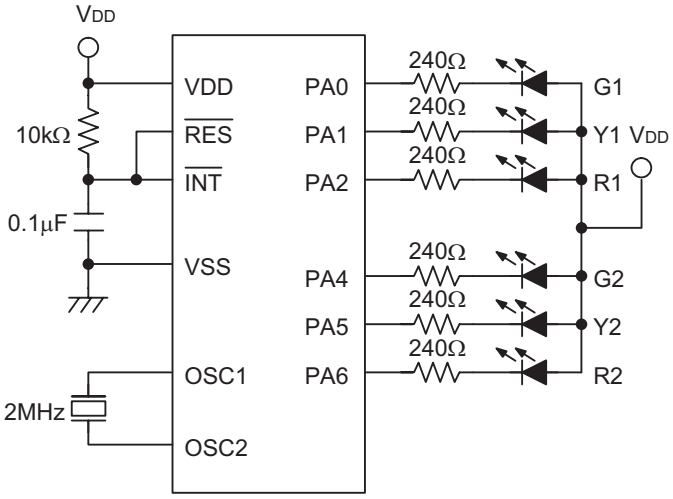
掩膜选项为默认值。

交通灯

这个应用使用红、绿和黄色发光二极管模拟十字路口的交通灯。开始时，R1 和 G2 点亮。一段延时后，绿灯闪烁，接着黄灯亮。再过一段延时，R2 和 G1 点亮。如此周而复始的循环，但是应用中的红绿灯点亮时间和闪烁时间是可以编程改变的。

→ **电路设计**

此电路使用两个端口部分 PA0~PA2 和 PA4~PA6，每个代表十字路口上一条道路的一组交通灯。电路使用解释独立于程序内容部分。更多的硬件信息可参见电路图。



HT48C10-1

→ 程序

```
//Traffic.c
//
//Body: HT48C10-1
//Mask option
//All the mask options use the default value.

#include <ht48c10-1.h>
const unsigned char table[16]={
0x14, 0x4, 0x14, 0x4, 0x14, 0x4, 0x14, 0x24,
0x41, 0x40, 0x41, 0x40,0x41, 0x40, 0x41, 0x42 };

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
     _intc = 0;
     _tmrc = 0;
     _tmr = 0;
     _pac = 0xff; //input mode
     _pbc = 0xff;
     _pcc = 0xff;
}
```

```
//a long time delay
void mydelay(unsigned int times){
    while(times--) _delay(65000);
}

void main(){
    unsigned char i, j, idx;

    safeguard_init();

    _pac = 0; //set port A to output port
    _pa = 0; //zero port A (all light on)
    while(1) {
       idx = 0;
       for(i=0; i!=2; i++) {
           _pa = table[idx];
           idx++;
           mydelay(8);
           for(j=0; j!=6; j++) {
                 _pa = table[idx];
                 idx++;
                 mydelay(1);
           }
           _pa = table[idx];
           idx++;
           mydelay(4);
       }
    }
}
```

→ 掩膜选项

掩膜选项为默认值。

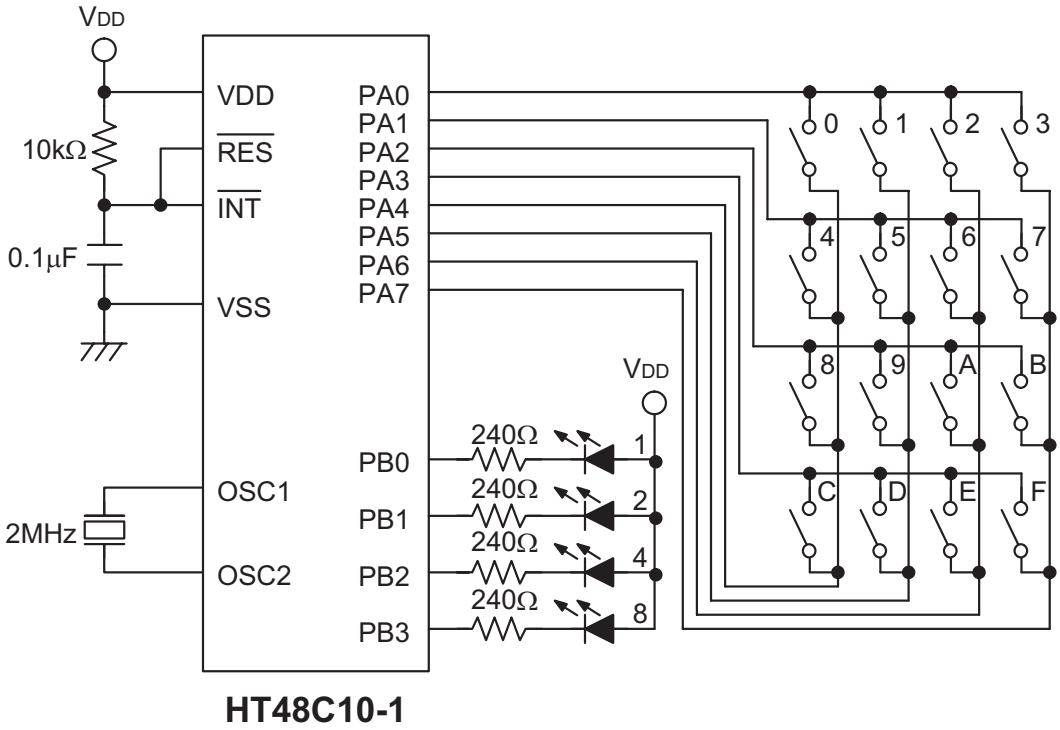
键盘扫描

此应用使用 4x4 的键盘矩阵，提供总共 16 个按键，每个按键表示单个的十六进制值，键值如图所示。 程序对键盘矩阵进行扫描以检测哪个键被按下，若检测到便会在 LED 上显示相应的十六进制数。电路中共有 4 个 LED，所以显示的数值可以从 0000 到 1111。在键扫过程中，若有两键同时按下，只有第一个被扫描到的按键会被检测到并且显示出来。使用这种方法，8 根逻辑线可以控制 16 个赋值开关

→ 电路设计

PA0~PA3 被用作输出，PA4~PA7 被用作输入，一起组成一个 4x4 矩阵. 注意在项目创建时，掩膜选项中的 PA/PA 应该选择带上拉而 BZ/BZB 应该选择ì All Disableî. 程序检测哪个键被按下而每个键的键值在一个待查表格中定义。

PB0~PB3 定义为输出，表示 4 位十六进制码，可以给出 16 种不同数值，每个数值代表一个按键。



→ 程序

```
//Keyboard.c
//
//Body: HT48C10-1
//Mask option
//BZ/BZB : All Disable
//the others use the default value

#include <ht48c10-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
```

```
    _intc = 0;
    _tmrc = 0;
    _tmr = 0;
    _pac = 0xff; //input mode
    _pbc = 0xff;
    _pcc = 0xff;
}

const unsigned char led_code[16]=
    {0xff, 0xfe, 0xfd, 0xfc, 0xfb, 0xfa, 0xf9, 0xf8,
     0xf7, 0xf6, 0xf5, 0xf4, 0xf3, 0xf2, 0xf1, 0xf0};
const unsigned char scan[4] = {0xfe, 0xfd, 0xfb, 0xf7};

//return the row number of the pressed key
unsigned char wait_key_pressed(){
    unsigned char i;
    i=0;
    while(1){
       _pac = scan[i];  //output scan code to port A
       if ((~_pa) & (unsigned char)0xf0){ //key pressed
           _delay(2000); //debounce
           //after debounce, if the key is still pressed
           //we claim it a key pressed, otherwise ignore it
           if ((~_pa) & (unsigned char)0xf0)
                 return i; //row i, key pressed
       }
       i++;
       if (i > 3) i = 0;
    }
}

// return the column number of the pressed key
unsigned char wait_key_released(){
    unsigned char i;
    unsigned char key;

    key = _pa; //keep the pressed key

    // wait until key released
    while((~_pa) & (unsigned char)0xf0);

    //find out which column key pressed
    //no debouce needed
    for(i=0; i<4; i++)
       if ((~key) & (0x10<<i))
           break; //column i, key pressed
    return i;
}

unsigned char get_key(){
    unsigned char row, col;
    row = wait_key_pressed();
    col = wait_key_released();
    return (row << 2) + col;
}
```

```
void main(){
    unsigned char index;

    safeguard_init();

    _pac = 0xff; //set port A as input port
    _pbc = 0x00; //set port B as output port
    _pa = 0;     //zero port A
    _pb = 0xff;  //off LEDs

    while(1){
       index = get_key();
       //the key value won't be displayed until
       // the key is released
       _pb = led_code[index];
    }
}
```

→ 掩膜选项

BZ/BZB 掩膜选项选择 *All* Disable, 其余选择默认值。

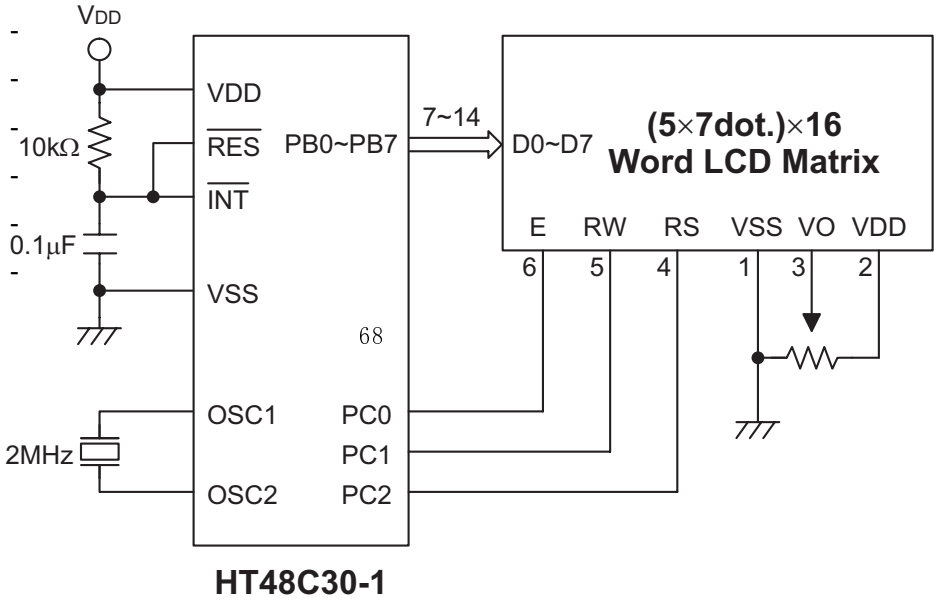
LCM

此应用示范了 8 位单片机与液晶显示 DV16100NRB 的联合使用。其中一个内部的 Hitachi HD44780 器件驱动和控制 LCM。应用中只需考虑产生正确的单片机机信号以满足 LCM 的时序需求。如果需要更多的时序和指令信息，则先要查看 LCM 生产商提供的数据。

LCM 的运行模式有 4 位和 8 位。如果使用 4 位模式运行，要传送一个字符或指令给模块则需要两次传送动作才能完成。如果使用 8 位模式运行则只需一次，不过需要多加 4 根 I/O 线。

→ 电路设计

PB0~PB7 设置为 I/O 位，PC0~PC2 作为 LCM 控制线需要设置为输出。这些都可根据用户要求自行设置。



-

-

-

-

-

-

-

→ 程序

```
//Lcm.c
//
//Body: HT48C30-1
//Mask option
//BZ/BZB : All Disable
//the others use the default value

#include <ht48c30-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmrc = 0;
       _tmr = 0;
       _pac = 0xff; //input mode
       _pbc = 0xff;
       _pcc = 0xff;
}

//#define FOUR_BIT
//#define ONE_LINE

//for DV-16100NRB
// port B : LCM data port
// port C : LCM control port
#define LCM_CLS        0x1
#define CURSOR_HOME    0x2
#define CURSOR_SR      0x14
#define CURSOR_SL      0x10
```

```
#define INCDD_CG_SHF_C  0x6
#define TURN_ON_DISP    0xf
#define LCD_ON_CSR_OFF  0xc

#define LCM_DATA        _pb
#define LCM_DATA_CTRL   _pbc
#define LCM_CTRL        _pc
#define LCM_CTRL_CTRL   _pcc

#define LCM_CTRL_E      _pc0
#define LCM_CTRL_RW     _pc1
#define LCM_CTRL_RS     _pc2

#define WRITE(a)        { \
LCM_DATA = (a); \
LCM_CTRL_E = 1; \
LCM_CTRL_E = 0;}

const unsigned char msg[16] = "HOLTEK 8 bit MCU";

void mydelay(unsigned char ct);
void LCM_initialize();
void send_cmd(unsigned char);
void write_char(unsigned char);
void busy_check(void);
void lcm_delay(void);

void main(){
       unsigned int i;

       safeguard_init();
       LCM_initialize();

       while(1){
              send_cmd(LCM_CLS);
              mydelay(2);
              send_cmd(CURSOR_HOME);
              for(i = 0; i<sizeof(msg); i++){
                     if (i == 8)
                            send_cmd(0xc0); //move cursor
                                            //to 2nd line
                     write_char(msg[i]);   //(1st line:00h~,
                                            // 2nd line:40h~)
              }
              send_cmd(LCD_ON_CSR_OFF);
              mydelay(5);
       }
}

void mydelay(unsigned char ct){
       while(ct--) _delay(65535);
}

void LCM_initialize(){
       LCM_DATA_CTRL = 0;//setup LCM data port as output port
       LCM_CTRL_CTRL = 0;//setup LCM control port as output port
```

```
        LCM_DATA = 0;     //clear LCM data port
        LCM_CTRL = 0;     //clear LCM control port

#ifdef FOUR_BIT
        WRITE(0x20); //4 bit mode
#else
        WRITE(0x30); //8 bit mode
#endif

//According to the data for the HD44780, there needs to be at
//least 4.5 ms delay between each program.
        mydelay(1);

#ifdef FOUR_BIT
#ifdef ONE_LINE
        WRITE(0x20); //4-bit 1-line
#else
        WRITE(0x28); //4-bit 2-line
#endif
#else
#ifdef ONE_LINE
        WRITE(0x30); //8-bit 1-line
#else
        WRITE(0x38); //8-bit 2-line
#endif
#endif

#ifdef FOUR_BIT
        WRITE(0x80); //4-bit high nibble (2nd pass)
#endif
        send_cmd(LCM_CLS);          //clean display
        send_cmd(TURN_ON_DISP);     //turn on display
        send_cmd(INCDD_CG_SHF_C);   //auto increment mode
                        //cursor left and DD RAM address+1
}


// send command to LCM
void send_cmd(unsigned char c){
#ifdef FOUR_BIT
        unsigned char tmp;
        tmp = c << 4;
        c &= (unsigned char)0xf0;
#endif
        busy_check();
        LCM_DATA = c;
        LCM_CTRL_RW = 0;
        LCM_CTRL_RS = 0;
        LCM_CTRL_E = 1;
        LCM_CTRL_E = 0;
#ifdef FOUR_BIT
        WRITE(tmp);
#endif
}

// write character to LCM
```

```
void write_char(unsigned char c){
#ifdef FOUR_BIT
       unsigned char tmp;
       tmp = c<<4;
       c &= (unsigned char)0xf0;
#endif
       busy_check();
       LCM_DATA = c;
       LCM_CTRL_RW = 0;
       LCM_CTRL_RS = 1;
       LCM_CTRL_E = 1;
       LCM_CTRL_E = 0;
#ifdef FOUR_BIT
       WRITE(tmp);
#endif
}

// Wait until the busy flag is not busy
void busy_check(void){
       unsigned char val, tmp;
       do{
              LCM_CTRL_E = 0;
              LCM_DATA_CTRL = 0xff;
              LCM_CTRL_RS = 0;
              LCM_CTRL_RW = 1;
              LCM_CTRL_E = 1;
              val = LCM_DATA;
              LCM_CTRL_E = 0;
#ifdef FOUR_BIT
              tmp = val & (unsigned char)0xf0;//4-bit high
nibble
              LCM_CTRL_E = 1;   //pulse high
              val = LCM_DATA;    //4-bit low nibble (2nd pass)
              LCM_CTRL_E = 0;   //pulse low
              val = (val>>4) | tmp; //combine 2 pass
#endif
       }while(val & (unsigned char)0x80);
       LCM_CTRL_RW = 0;
       LCM_DATA_CTRL = 0;   //LCM not busy, then set LCM data
                            //bus to input port
}
```

→ 掩膜选项

BZ/BZB 掩膜选项选择 *All Disable*, 其余选择默认值。

I/O 端口的串行应用

此应用示范了模拟串行操作的代码。此应用可被用于开发简单的串行端口应用的基础，比如 8 位通讯、无奇偶校验、单个停止位的应用。

→ 程序

```
//Serial.c
//
//Body: HT48C70-1
//Mask option
//WDT : Disable
//the others use the default value

#include <ht48c70-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmr0c = 0;
       _tmr0h = 0;
       _tmr0l = 0;
       _tmr1c = 0;
       _tmr1h = 0;
       _tmr1l = 0;
       _pac = 0xff;
       _pbc = 0xff;
       _pcc = 0xff;
       _pdc = 0xff;
       _pec = 0xff;
       _pfc = 0xff;
       _pgc = 0xff;
}

#define tx   _pa3             //transmit pin
#define rx   _pa2             //receive pin
#define _pac3 _13_3
#define _pac2 _13_2

unsigned char data;

void transmit(unsigned char);
void receive(unsigned char *);


//system frequency: 4MHz
//#define T 38 //baudrate 19200 = 4M/4/(T+14) => T = 38
#define T 90 //baudrate 9600 = 4M/4/(T+14) => T = 90
//#define T 194 //baudrate 4800 = 4M/4/(T+14) => T = 194
//#define T 402 //baudrate 2400 = 4M/4/(T+14) => T = 402

void main(){
       safeguard_init();
```

```
        _pac2 = 1;      //set receive pin to input mode
        _pac3 = 0;      //set transmit pin to output mode

        while(1){
                receive(&data);
                transmit(data);
        }
}

void transmit(unsigned char val){
        unsigned char i;

        tx = 0;
        for(i=0; i<8; i++){
                _delay(T);
                if (val & 1) tx = 1;
                else tx = 0;
                val >>= 1;
        }
        _delay(T);
        tx = 1;
        _delay(T);
}

void receive(unsigned char *val){
        unsigned char i, v;

        v = 0;
        while(rx); //wait start bit
        for(i=0; i<8; i++){
                _delay(T);
                if (rx) v |= (unsigned char)0x80;
                v >>= 1;
        }
        _delay(T);//skip stop bit
        _delay(T);
        *val = v;
}
```

→ 掩膜选项

WDT 掩膜选项选择 *Disable*, 其余选择默认值。

中断和定时/计数器的应用

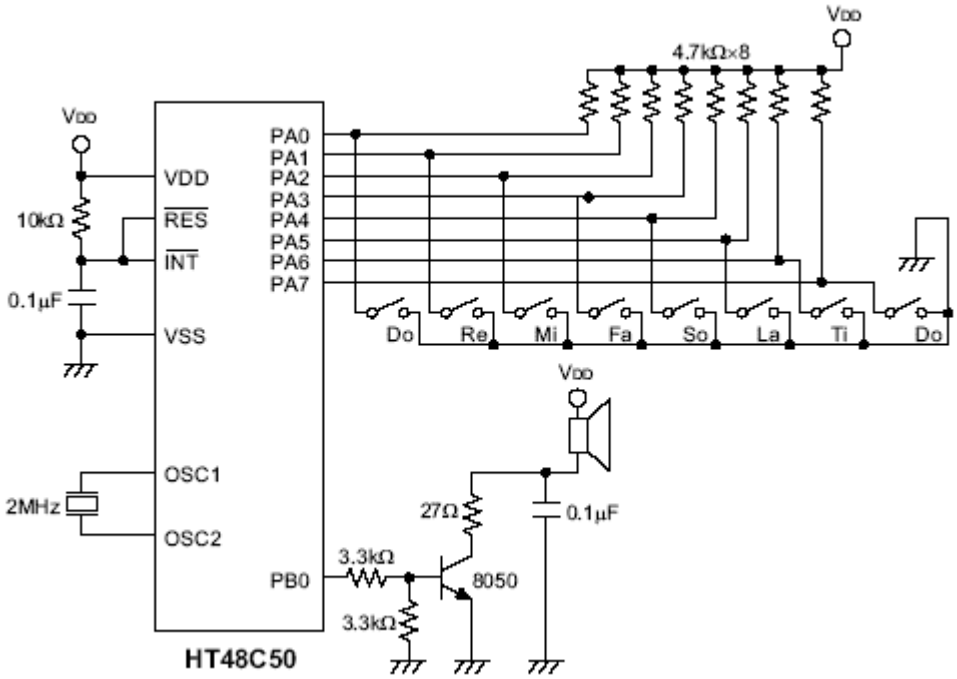
电子钢琴

此范例示范了怎样实现一个扫描键盘并且从按下的键产生对应的定义的声音

频率。每当一个按键按下相应的频率值将被写入定时/计数器寄存器。当计数器计数到最大值会产生一个内部中断并且运行中断程序。此时定时/计数器的值会重新载入并且继续计数。这样，对定时/计数器写入不同的值将会产生不同的频率。在内部的中断程序中改变输出端口状态使得对应的引脚产生需要的频率，从而得到想要的音符。加入合适的扩音器和喇叭整个系统就完整了。软件中最重要的是使用定时/计数器作为计数工具控制输出频率，输出频率值需要计算。

→ 电路设计

PA0~PA7 为输出引脚, 每个引脚都通过串联一个 240Ω的电阻控制一个 LED。通过右移和左移运算将点亮的 LED 由左至右和由右至左移动。具体电路参见电路图。



→ 程序

```
//Piano.c
//
//Body: HT48C50-1
//Mask option
//BZ/BZB : All Disable
//the others use the default value
```

```
#include <ht48c50-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmr0c = 0;
       _tmr0 = 0;
       _tmr1c = 0;
       _tmr1h = 0;
       _tmr1l = 0;
       _pac = 0xff;
       _pbc = 0xff;
       _pcc = 0xff;
       _pdc = 0xff;
}


#define _tmr1c4 _11_4  //timer1 enable bit


const unsigned char frq[16] = {
0x21, 0xfe, 0x58, 0xfe, 0x84, 0xfe, 0x99, 0xfe,
0xc1, 0xfe, 0xe3, 0xfe, 0x2, 0xff, 0x11, 0xff};

unsigned char frq_idx;

void initial();
void wait_key_press();
void wait_key_release();
void start_sound();
void stop_sound();

void main(){

       safeguard_init();
       initial();

       while(1){
              wait_key_press();
              start_sound();
              wait_key_release();
              stop_sound();
       }
}

void wait_key_press(){
       unsigned char i, key;
```

```
        key = 0;
        while(!key)
                key = ~_pa;

        for(i=0; i<8; i++){
                if (key & 0x1){
                        frq idx = i << 1;
                        break;
                }
                key >>= 1;
        }
}

void wait_key_release(){
        unsigned char key;
        key = 1;
        while(key)
                key = ~_pa;
}

void start_sound(){
        _intc = 9;              //enable timer1
        _tmr1c = 0x80;          //timer mode
        _tmr1l = frq[frq_idx];  //load sound freq.
        _tmr1h = frq[frq_idx+1];
        _tmr1c4 = 1;            //start timer1
}

void stop_sound(){
        _tmr1c4 = 0;            //stop timer1
        _pb = 0;
}

void isr_c(){                   // timer1
        _pb = ~_pb;             // generate square wave
}

void initial(){
        _pac = 0xff;    //set port A to input port
        _pbc = 0;       //set port B to output port
        _pb = 0;
}
```

→ 掩膜选项

BZ/BZB 掩膜选项选择 *All Disable*, 其余选择默认值。

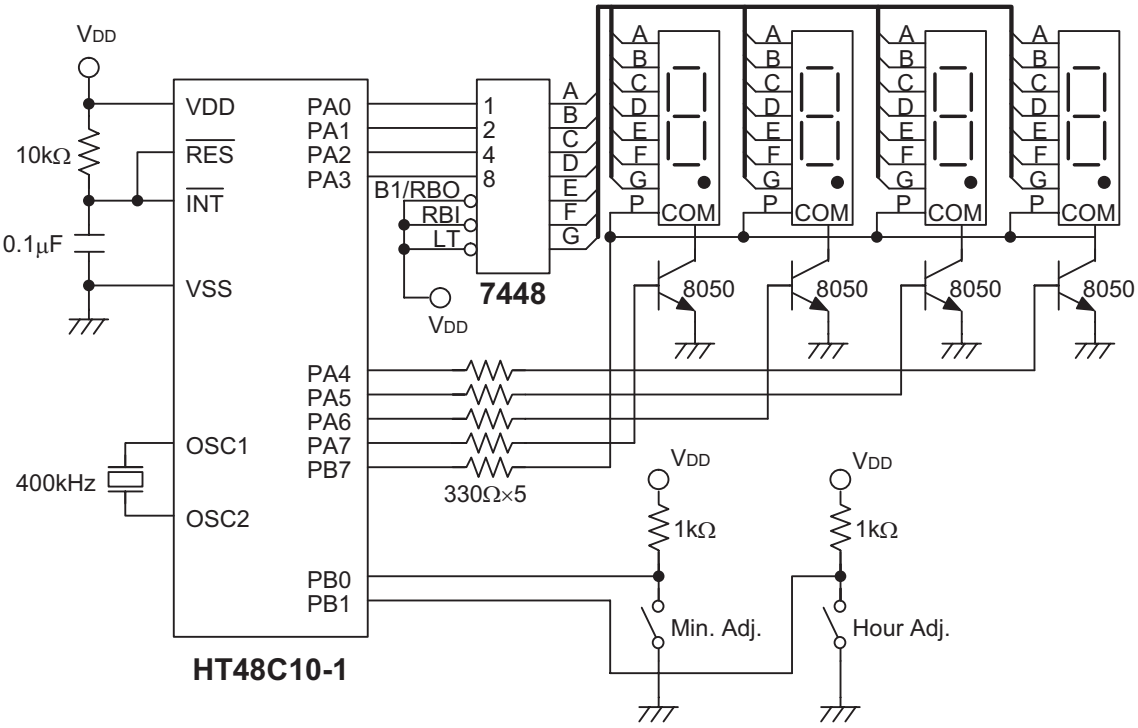
时钟

此应用示范了使用 16 位定时计数器产生内部中断以实现计时功能。这个应用依靠系统时钟频率作为计时的基准。此处所示的应用使用了 400KHz 的系统时

钟，通过内部除四分频产生 100KHz 的定时/计数器时钟。对于一个 16 位的计数器最大计数值为 65536，这将每隔 0.65536 秒产生一个内部中断。但是时钟需要 1 秒作为基本时间单元。因此定时/计数器被设置成记录 0.5 秒的基准时间，而数取两次中断级可得到 1 秒的基准时间。这里的应用使用 4 个 7 段码显示 24 小时制的时间，显示小时和分钟。而小时和分钟的调节由两个键来控制。

→ 电路设计

PA0~PA7 为输出，PA0~PA3 用于显示数据。PA4~PA7 提供扫描输入给用于段节显示的三极管。这些引脚将逐个扫描单个显示。PB0 和 PB1 设置为输入，给调节小时和分钟的开关使用。



→ 程序

```
//Clock.c
//
//Body: HT48C50-1
//Mask option
//BZ/BZB : All Disable
//SysFreq: 400KHz
//the others use the default value
```

```
#include <ht48c50-1.h>

#define _tmr1c4 _11_4  //timer4 enable bit
#define min_adj_button  _pb0
#define hour_adj_button _pb1

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmr0c = 0;
       _tmr0 = 0;
       _tmr1c = 0;
       _tmr1h = 0;
       _tmr1l = 0;
       _pac = 0xff;
       _pbc = 0xff;
       _pcc = 0xff;
       _pdc = 0xff;
}

void initial();
void check_time();
void show_clock();
unsigned char min_adj_pressed();
unsigned char hour_adj_pressed();
void min_adjust();
void hour_adjust();
void arrange_hour();
void set_timer();

unsigned char half_second;
unsigned char min_l, min_h;
unsigned char hour_l, hour_h;

void main(){

       safeguard_init();
       initial();

       while(1){
              check_time();
              show_clock();
              if (min_adj_pressed())  min_adjust();
              if (hour_adj_pressed()) hour_adjust();
       }
}
```

```
void isr_c(){          //timer1
       half_second++;
       _pb = ~_pb;     //flash 'dot' every 0.5 second
}

void initial(){
       _pac = 0;       //set port A to output port
       _pbc = 0x7f;    //set port B to input port exclude pb7
       _pb = 0;
       _pa = 0;

       min_l = 0;
       min_h = 0;
       hour_l = 0;
       hour_h = 0;
       half_second = 0;

       _intc = 0x9;    //enable timer1
       _tmr1c = 0x80;  //timer1 mode (internal clock)
       set_timer();
}

//check if the min_adj_button is pressed or not
//return 1: if the min_adj_button is pressed
//       0: otherwise
unsigned char min_adj_pressed(){
       if (min_adj_button == 0){//pressed
              _delay(2000); //debounce
              if (min_adj_button == 0)
                     return 1; //still pressed, recognize it
       }
       return 0;
}

//check if the hour_adj_button is pressed or not
//return 1: if the hour_adj_button is pressed
//       0: otherwise
unsigned char hour_adj_pressed(){
       if (hour_adj_button == 0){//pressed
              _delay(2000); //debounce
              if (hour_adj_button == 0)
                     return 1; //still pressed, recognize it
       }
       return 0;
}

void check_time(){
       if (half_second >= 120){
              half_second -= 120;
              min_l++;
              if (min_l >= 10){
                     min_l = 0;
                     min_h++;
                     if (min_h >= 6){
                            min_h = 0;
                            hour_l++;
```

```
                        arrange_hour();
                    }
                }
            }
        }

//This function is to arrange the hour value
void arrange_hour(){
        if (hour_h == 2 && hour_l == 4){
                hour_h = 0;
                hour_l = 0;
        }
        else if (hour_l == 10){
                hour_l = 0;
                hour_h++;
        }
}

void show_clock(){
        _pa = min_l | 0x10;
        _pa = min_h | 0x20;
        _pa = hour_l | 0x40;
        _pa = hour_h | 0x80;
}

//This function is to adjust the minute.
//The minute will increase 1 when the min_adj_button is
pressed .
//If the button is held longer than 1.5 seconds, the minute will
//increase 1 every 0.5 second
void min_adjust(){
        bit held_long_time = 0;


repeat_inc:
        min_l++;
        if (min_l >= 10){
                min_l = 0;
                min_h++;
                if (min_h >= 6) //don't care hour
                        min_h = 0;
        }
        half_second = 0;
        while(min_adj_button == 0){//while min_adj_button
                show_clock();          // is held
                if (!held_long_time){
                        if (half_second>2){//longer than 1.5 sec
                                held_long_time = 1; //set flag
                                goto repeat_inc; //increase minute
                        }
                        //less than 1.5 seconds, do nothing
                }
                else{
                        if (half_second)
                                goto repeat_inc; //inc 1, 0.5 sec
                        //less than 0.5 second, do nothing
```

```
                }
        }
        half_second = 0;
        set_timer();
}

//This function is to adjust the hour.
//The hour will increase 1 when the hour_adj_button is pressed .
//If the button is held longer than 1.5 seconds, the hour will
//increase 1 every 0.5 second
void hour_adjust(){
        bit held_long_time = 0;

repeat_inc:
        hour_l++;
        arrange_hour();
        half_second = 0;
        while(hour_adj_button == 0){
                show_clock();
                if (!held_long_time){
                        if (half_second>2){//longer than 1.5 sec
                                held_long_time = 1; //set flag
                                goto repeat_inc;//increase hour
                        }
                        //less than 1.5 seconds, do nothing
                }
                else{
                        if (half_second)
                              goto repeat_inc;//inc 1, 0.5 sec
                        //less than 0.5 second, do nothing
                }
        }
        half_second = 0;
        set_timer();
}

void set_timer(){
        _tmr1c4 = 0;
        _tmr1l = 0xb0;
        _tmr1h = 0x3c;
        _tmr1c4 = 1;    //start timer1
}
```