C Compiler V3 使用手册

版本：V2.00　日期：2024-03-22

www.holtek.com

用户须知

所有文档均会过时，本文档也不例外。Holtek 的工具和文档将不断演变以满足客户的需求，因此实际使用中有些对话框和工具说明可能与本文档所述之内容有所不同。请访问我们的网站 (http://www.holtek.com.tw/zh/mcu_tools_users_guide) 获取最新文档。

目录

前言

本手册主要讲述了 C 语言的基础语言，再以此为基础，进而讲述 C compiler V3 的语法结构和其优化功能，帮助程序员快速使用 C compiler V3 开发应用程序。

C compiler V3 是由 GCC 4.6.2 以上版本移植过来的，除后端输出，其余部份都可参考 GCC 与机器无关的相关使用手册。

这里假定读者已具备如下基本素质：

- 知道如何编写 C 程序
- 已经阅读并理解所使用单片机的数据手册

第一章 C 语言基础知识

本章将由浅到深的概括 C 语言的基础语法及结构特点，方便后面学习 C compiler V3，由于受限于单片机的硬件结构，因此本章的描述基于标准 C 语言，兼容 C compiler V3 之语法。

主要包含如下内容：

- 数据类型、运算符与表达式
- 函数
- 数组与指针
- 结构体、联合体与枚举
- 预处理
- 流程控制
- 作用域

1.1 数据类型、运算符与表达式

1.1.1 C 的数据类型

数据类型确定了变量在内存中占用的存储单元，所以在声明变量时首先必须要确定变量的类型，数据类型可以分为基本数据类型、构造数据类型、指针类型 (Pointer) 和空类型 (void)，基本数据类型有整型、字符型、浮点型，构造类型则有数组、结构体、共享体和枚举，利用这些构造类型可以构造出所需要的数据结构。

列举基本数据类型 (C compiler V3) 如表 1-1-1：

数据类型	占用空间 (bit)	范围
bit[1]	1	0, 1
_Bool[2]	8	0, 1
char [3]	8	-128~127
unsigned char	8	0~255
short	16	-32 768~32 767
unsigned short	16	0~65535
int	16	-32 768~32 767
unsigned int	16	0~65535
long	32	-2147483648~2147483647
unsigned long	32	0~4294967295
long long	32	-2147483648~2147483647
unsigned long long	32	0~4294967295
float [4]	24	-3.4E+038~3.4E+038
double [5]	32	-3.4E+038~3.4E+038
long double [5]	32	-3.4E+038~3.4E+038

表 1-1-1 基本数据类型 (C compiler V3)

注：[1]、bit 类型，最低位有效。比如 bit a = 4；则 a = 0。
[2]、_Bool 类型，当值非 0 时为 1，当值为 0 时则为 0。比如 _Bool a = 4；则 a = 1。
[3]、基本数据如果没有 unsigned 修饰，则默认为 signed，下同。

[4]、C compiler V3 的 float 型为 24 位，只有 4~5 位精度 (V3.20 以上版本支持)

符号 sign	指数 e	尾数 m
23	22~15	14~8 7~0

[5]、double 和 long double 为 IEEE 754 32-bit 格式，有 6~7 位精度 (V3.20 以上版本支持)

符号 sign	指数 e	尾数 m
31	30~23	22~16 15~8 7~0

1.1.2 常量与变量

在程序运行过程中，常数是其值不能被改变的量，与之对应的是变量，变量是在内存中具有特定属性的一个存储单元，它用来存放数据，根据所需要存放数据的数据类型定义相应的变量。

1.1.2.1 数字常量、字符串常量、const 常量

- 数字常量：C Compiler V3 支持指定十六进制 (0x) 和八进制 (0) 值的标准前缀，另外还支持用前缀 0b 来指定二进制值。例如，数值 237 可以表示为二进制常数 0b11101101。
- 字符串常量：一般用“”引起来的符号串，如“abc”。
- const 常量：C Compiler V3 支持最大 64Page 的 const 常量，且支持 const 数组，指向 const 的指针等。

Example：

```
const unsigned char TABLE[8]={1,2,3,4,5,6,7,8}
unsigned char Test[10];
void TEST_Const(unsigned char *data, unsigned char counter)
{
   unsigned char i, T_counter;
   T_counter = counter;
   for(i=0; i< T_counter; i++)
   {
     Test[i] = *data++;
   }
}
void main()
{
   TEST_Const(TABLE,8);
}
```

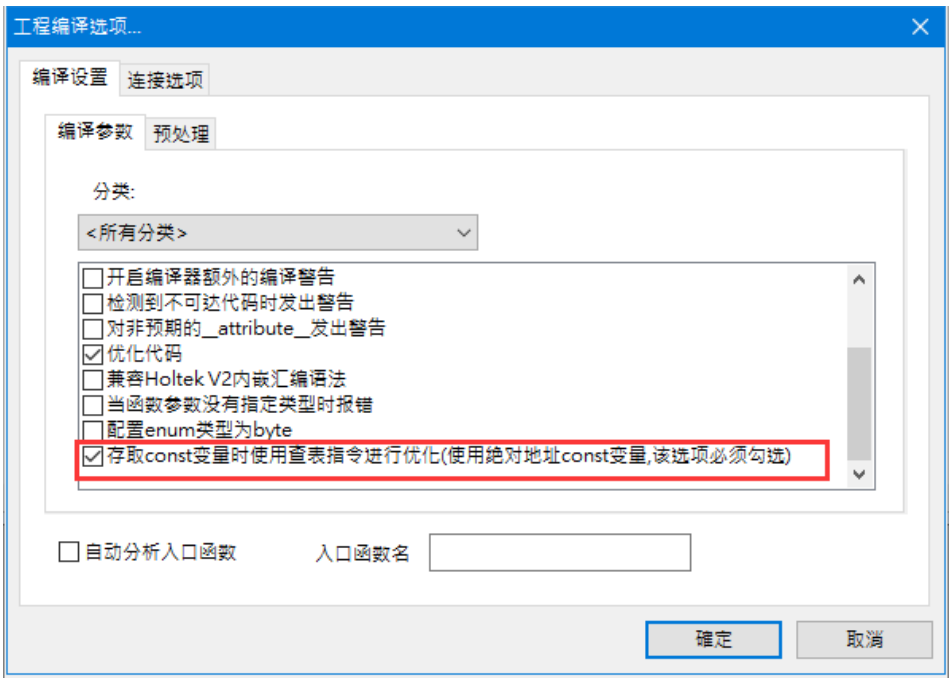
对有扩展指令的 MCU (如 HT66F70A) 或具 TBHP 寄存器且 PROM 宽度为 16bit，V3 Compiler(v3.20 以上版本) 支持指定地址的 const 变量，其语法如下：

```
const type  __attribute__ ((at(addr)))  var[] = {1,2,3,4,5};
```

比如：

```
const char __attribute__ ((at(0x123))) ci[5] = {1,2,3,4,5};
```

当 MCU 无扩展指令时，需勾选以下参数：



如此，ci 在 ROM 中的存放结果如下：

地址	0x123	0x124	0x125	…
内容	0x0201	0x0403	0x0005	…

1.1.2.2 变量及其定义

变量是在程序运行时，其值可以变化的，每个变量都应该有一个名字，以便被引用，并区分大小写，C 语言规定，所有的变量都必须先宣告，后使用，变量在定义时必指定数据类型，这样编译时才可为其分配对应的存储单元。如 int a;。

标识符是用来标识变量、常量、函数、类型等的字符系列，C 语言规定标识符只能由字母、数字、下划线构成，且必须以字母和下划线作为起始符。

1.1.2.3 变量的存储方式

在 C 语言中每个变量和函数都有两个属性：数据类型和数据的存储方式，存储方式可分为 2 种，静态存储和动态存储，具体又包含 4 种，自动的 (auto)、静态的 (static)、寄存器的 (register)、外部的 (extern)。

1、auto：函数中的局部变量，如果不专门声明为 static 的存储方式，则默认为 auto，所以在函数内 auto char a 与 char a 是等价的。

2、static：可分为全局静态存储和局部静态存储，全局变量加了 static 后，变量只能在本文件中引用，局部静态存储则是局部变量的值在函数调用结束后不消失而保留原值，在下一次调用该函数时，该变量已经有值了。

3、register：上述两种变量是存放在内存中的，而 register 则是把变量存放在寄存器中，基于单片机的特殊情况，这里不展开叙述。

4、extern：使用另一个文件中定义的变量，表示该变量是一个已经在外部定义过的变量，只要加上 extern 就可以使用该变量了，后面有专门的讲述。

5、volatile：一个类型修饰符 (type specifier)。它是被设计用来修饰被不同程序访问和修改的变量，使用 volatile 修饰的变量，不会因编译程序的优化而被省去。

建议定义成 volatile 的变量：特殊寄存器，中断函数使用到的变量，为某些特殊用途的代码定义的变量 (比如 delay 功能)。

其它一般变量不建议定义成 volatile，这样会大大降低编译程序的优化功能。

代码清单 1.1：

```
File1.c
int cpv;

File2.c
extern cpv;                          // 引用 File1.c 中的 cpv 变量
int c;
int statictest()
{
   cpv = 5;
   static int k = 26;                // 静态变量
   auto int p = 0;                   //auto 可省略
   k++;
   p++;
   cpv++;
   return (k + p + cpv);
}

void main()
{
   c = statictest();                 //c = 34
   c = statictest();                 //c = 35
}
```

两次运行的结果是 34，35

1.1.3 C 语言运算简介

C 语言的运算十分丰富，主要包括算术运算，逻辑运算、位运算、赋值运算、条件运算、逗号运算等，各种运算及其之间的优先级见附录 B。

1.1.3.1 类型转换

类型转换规则：

- 混合类型的算术运算
  - ♦ 小于 int 类型的转换为 int 类型
  - ♦ 小类型向大类型转换 (转换过程见图 2_1_1)
- 不同类型之间的赋值
  - ♦ 以赋值语句左边的类型为转换后类型
- 函数参数 / 返回值的传递
  - ♦ 以参数 / 返回值的类型为转换后类型

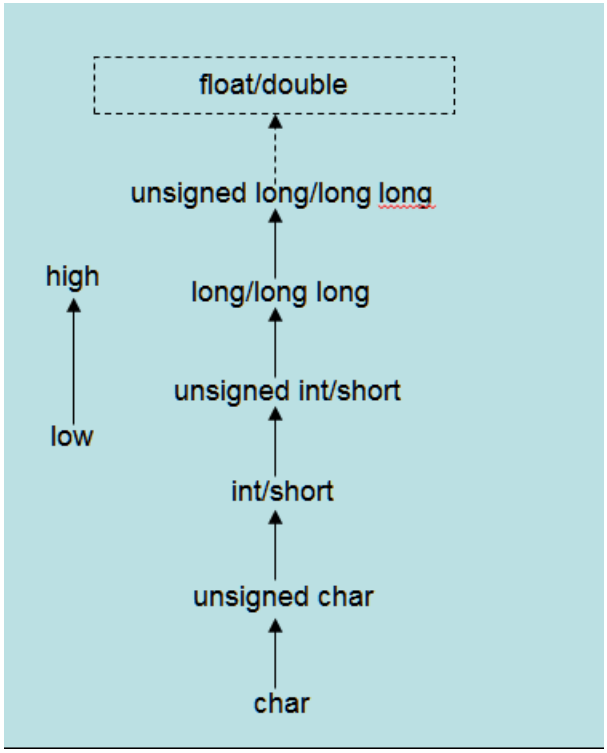


图 2_1_1

类型转换示例：

int a = 20000;
int b = 20000;
void main(void)
{
long c = a+b;
}
	char a = 100;
	char b = 100;
	void main(void)
	{
	int c = a+b;
	}

结果：C = -25536； 说明：a 与 b 都是 int 类型，使用 int 计算，结果为 40000，超出 int 的范围，所以结果为 -25536。 解决方式： long c = (long)a+b; 结果：c =40000；	结果：c = 200； 说明：a 与 b 都 <int，提升为 int 计算，结果为 200

1.1.3.2 逻辑运算

逻辑运算和关系运算的结果只有两个：真、假，如果运算出来的结果为真，则用逻辑 1 来表示，反之则为逻辑 0，逻辑运算就是把各关系运算或其它逻辑量进行与 (&&)、或 (||)、非 (!) 的运算。

逻辑运算具有短路性。

代码清单 1.2：

```
char a, b, c, d, e, f, g, h;
void main()
{
   a = 0x41;
```

```
    b = 0x31;
    e, f;
    c = 0xaa, d = 0x55, g = 0x5a, h = 0xa5;
    if((c = a > b) || (d = a)){           // 逻辑或的短路功能
           e = 0x18;
    }
    else{
           e = 0x81;
    }

    if((g = a < b) && (h = a)){           // 逻辑与的短路功能
           f = 0x18;
    }
    else{
           f = 0x81;
    }
}
```

运算结果：a = 0x41, b = 0x31, c = 0x01, d = 0x55, e = 0x18，f = 0x81, g = 0x00, h = 0xa5，d 和 h 没有被赋值。

非运算 (!) 的特殊应用：!!(0xXX)，当 0xXX 不为 0 时，两次取非运算的结果则为 1。

1.1.3.3 位运算

C 语言中两个很具魅力的地方，一个是指针，另一个是位运算，本小节讲述的便是 C 语言中的位运算，C 语言中位运算共有 6 种，按位与 (&)、按位或 (|)、按位异或 (^)、取反 (~)、左移 (<<) 和右移 (>>)，巧妙利用位运算可以简化很多运算时间。常见应用及操作：

1、把小写字母变为大写字母，清位：‘a’ & 0xDF，结果为‘A’

2、把大写字母变为小写字母，置位：‘A’ | 0x20，结果为‘a’

3、对某位取反， 某个位与 1 异或即为取反 (第 1 位取反)：0xFF ^ 0x01，，运算的结果为 0xFE

4、部分乘法的化简，与 2 的 n 次方相乘，相当于左移 n 位，例如 0x02 乘以 4，0x02 << 2，这里的 2，表示 4 = 2 的‘2’次方，结果为 8

5、部分除法的化简，与 2 的 n 次方相除，相当于右移 n 位，例如 0x08 除以 4，0x08 >> 2，这里的 2，表示 4 = 2 的‘2’次方，结果为 2

6、部分求余的化简，与 2 的 n 次方求余，跟 2 的 n 次方 -1 与，如 15 跟 8 求余，相当于 15 & 7，这里的 7 是 8-1 = 7，结果为 8

7、其他乘法的化简，例如 0x08 * 7 = 0x08 * (8 - 1) = (0x08 << 3) - 0x08

8、循环移位，对一个 16 位的数循环左移 n 位，0xXX >> (16 - n) | 0xXX << n

9、循环移位，对一个 16 位的数循环右移 n 位，0xXX << (16 - n) | 0xXX >> n

1.2 函数

函数是相当于子程序，每一个函数都可用来实现一个特定的功能。

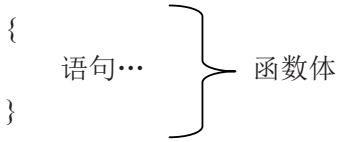
在程序开发的过程中，常使用的功能模块编写成函数，以减少重复编写程序段的工作量，函数的本质是一个标号，即一个地址，调用该函数，相当于跳转到这个地址去执行指令。

1.2.1 函数的声明与定义

函数声明时，不必先定义函数，只是做一个声明。声明的函数必须是存在的函数，如果使用库函数，则直接把头文件包含进来。如果是用户自定义的函数，则是声明函数将在该文件后面定义，在函数内不能再定义函数，即不能嵌套定义。

函数的定义包括返回值数据类型，函数名，参数列表和函数体，形式：

返回值数据类型 函数名 (参数列表)



例子 2：求两个数的较大者。

代码清单 1.3：

```
int max(int, int);                  // 函数声明
int a;
void main()
{
   a = max(10, 20);                 // 函数调用
   a++;
}
int max(int a, int b)               // 返回值类型 函数名 ( 参数列表 )
{
   return a > b ? a : b;
}
```

运算结果 a = 21

1.2.2 参数列表与返回值

参数列表，函数名之后，用括号括起来，在调用函数时要传递给函数的变量列表称为参数列表。

关于形参与实参的说明：

1、实参可以是常量、变量或表达式，但要求具有确定的值，在函数调用时，把值赋给形参。

2、函数的定义时，必须指定形参的数据类型。

3、实参与形参必须要兼容。

4、形参的最大特点是单向值传递，即只是把值传过来，而并没有改变实际参数的值。

5、在 C compiler V3 中，参数及函数局部变量的命名方式是 _funname_2[n]，比如：函数 fun 的变量命名为 _fun_2，n 则表示它共有多少个变量。

6、若返回值为 1byte，则存于 ACC，若为 2byte，则低字节存于 ra，高字节存于 rb，若为四 byte，由低字节到高字节分别存于 ra、rb、rc、rd。

7、由于 MCU 的限制 (没有堆栈)，参数和内部变量会占用 RAM 空间，而互不影响 (没有调用关系) 的函数变量可以共享同样的 RAM 空间。

例子 3：可以改变 a 的值吗？

代码清单 1.4：

```
int a;
void change(int b)
```

```
{
  b = 7;
}
void main()
{
  a = 15;
  change(a);
}
```

运算结果 a = 15

返回值是把运算的结果返回给该函数的调用者使用，void 类型的函数是否可以使用 return 呢？可以的，只是 return 后面不要加任何表达式。

1.2.3 函数的调用

1.2.3.1 函数的调用方式及过程

函数的调用方式有 3 种：

1、函数语句：只把函数调用作为一个语句，完成一定的功能，如 change(a);

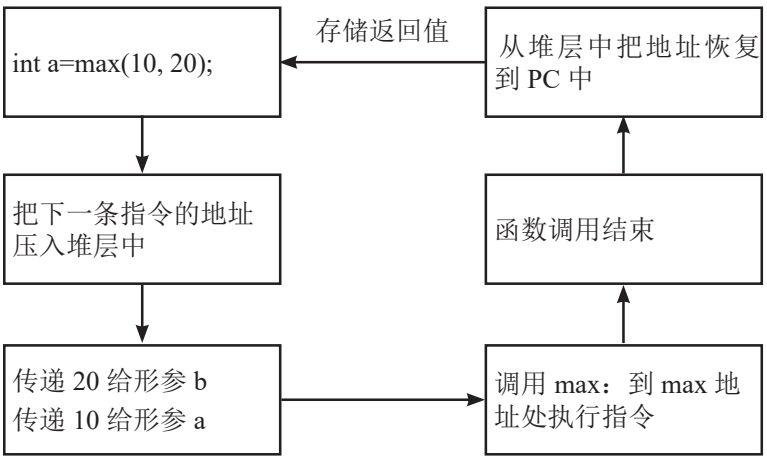
2、函数表达式：如 int a = 3*max(10, 20);

3、函数参数：如 int a = max(max(20, 30), 20);

函数的调用过程：

不同的编译程序其函数调用过程不尽相同，函数参数的传递的方向也不相同，在 C 语言中，形参是从右到左的传递，下面讲述 C compiler V3 下函数调用过程 (以例子 2 为例子)：

流程 1：



函数局部变量的地址分配：

1、HT8 MCU 没有 RAM stack, 所有函数的局部变量都是编译时 (link 阶段) 分配好的，占用 RAM 空间

2、Linker 根据函数调用关系分配局部变量，若 function1 与 function2 无调用关系，则 function1,function2 可以共用局部变量空间 , 有调用关系则无法共用。

3、Linker 为主程序跟中断各分配独立的局部变量空间，不会共用 (也不允许中断与主程序共同调用子函数)。

4、除了局部变量，编译器还会用到 8 个中间变量 ra~rh，因为没用通用寄存器，所以 ra~rh 也是分配在普通 RAM 里面，同样主程序跟中断的 ra~rh 也是独立分开分配。

1.2.3.2 嵌套调用

函数的嵌套调用是指在函数内调用别的函数，函数是可以嵌套调用的，但是不可以嵌套定义，而递归调用则是特殊的嵌套调用，是指一个函数内又直接或间接调用该函数本身。

1.2.3.3 外部函数的使用

使用外部函数有两种方式，一种是通过头文件包含，一种则是使用 extern 关键词。

1、头文件包含的方式：

例子 4：调用外部加，减函数。

代码清单 1.5：

```
File1.h
#ifndef _File1_H
#define _File1_H
typedef unsigned char u8;
u8 add(u8 a, u8 b);
u8 sub(u8 a, u8 b);
#endif
File1.c
#include "FILE1.H"
u8 add(u8 a, u8 b)
{
   return a + b;
}
u8 sub(u8 a, u8 b)
{
   return a - b;
}

File2.c
#include "FILE1.H"
u8 c, d;
void main()
{
   u8 a = 10;
   u8 b = 10;
   c = add(a, b);
   d = sub(c, b);
}
```

执行结果：a = 10, b = 10, c = 20, d = 10

2、extern 的方式：

例子 5：调用外部乘，除函数。

代码清单 1.6：

```
File1.c
typedef unsigned char u8;
u8 mul(u8 a, u8 b)
{
   return a * b;
}
u8 div(u8 a, u8 b)
{
   return a / b;
}
```

```
File2.c
typedef unsigned char u8;
extern u8 mul(u8 a, u8 b);
extern u8 div (u8 a, u8 b);
u8 c, d;
void main()
{
   u8 a = 10,b = 10;
   c = mul(a, b);
   d = div(a, b);
}
```

执行结果：a = 10, b = 10, c = 100, d = 1

1.2.3.4 内联函数

使用关键词 inline 修饰函数，则该函数为内联函数，内联函数直接把函数体的语句替换函数的调用。以节省函数调用时因保护现场的时间代价和使用堆层的空间代价。多次调用时会大量增加代码。

代码清单 1.7：

```
unsigned char max;
inline void getmax(unsigned char var1, unsigned char var2)
{
  max = var2 > var1 ? var2 : var1;
}

void main()
{
  getmax(23,32);
}
```

这里直接把 max = var2 > var1 ? var2 : var1; 这句代码代换了 getmax(23,32); 函数。

1.2.4 main 函数

main 函数是特殊的函数，程序运行时的入口函数，当初始化完单片机的环境之后，接下来就执行 main 函数中的语句。所以每个工程都应该有一个 main 函数，main 函数不需要参数及返回值。

1.2.5 标准函数库

C Compiler V3(v3.20 以上版本) 支持 math.h、string.h 等标准函数库，其使用说明详见 IDE/DOC--< 标准函数库使用手册 >。

1.3 数组与指针

数组是有序数据的集合，数组中的每一个函数都属于同一种数据类型，使用数组名和下标可以唯一确定数组中的一个元素。内存区中每个字节都有一个编号，这个编号称为地址，指针变量的值就存放这些编号。

1.3.1 数组的定义、初始化与使用

数组的定义：数据类型 数组名 [元素个数]，元素个数只能是直接常量和符号常量。如 unsigned char led_table[5] [1]。

数组的初始化方式：

1、定义一个数组，如 unsigned char led_table[5];[2] 然后逐个元素赋值。

2、定义的同时初始化，如 unsigned char led_table[5] = {0, 1, 2, 3, 4};

3、不指定个数的初始化，如 unsigned char led_table[] = {0, 1, 2, 3, 4}; 这时会确

定元素个数为 5 个。

4、部分初始化，如 unsigned char led_table[5] = {2, 1};[3]

注：[1]、数组下标由 0 开始，表示第一个元素，最后一个元素为个数 -1。

[2]、逐个赋值时，数组索引不能超过元素个数，如本处元素个数为 10，则下标最大为 9，超过会溢出。

[3]、led_table[0] = 2，led_table[1] = 1 其他没有被赋值的值为 0。

使用时，直接用数组名加索引号，如 led_table[2]，由于 C 语言不检查数组界限，所以在使用时要注意数组下标溢出的问题。

1.3.2 多维数组

有多个下标索引，定义并初始化时，最高维不用指定，例如：led_table[][4] = {{1, 2, 3},{},{1}}; 本例中第二维已经确定为 3，使用时，每个维数的下标都要指定。

1.3.3 字符串及其结束标志

字符串常量存放在 ROM 中，以‘\0’结束，例如，字符串 char c[] = “chip”，所以“chip”实际占用了 5 个字符空间。

如果一个字符数组，最后一个元素为‘\0’也可当字符串使用。

数组名的本质是数组在内存空间中的首地址，例如 led_table[5]，则 led_table 为 led_table[0] 的地址。

1.3.4 指针的概念

在程序中定义的变量，会在内存中分配相应的内存单元，而该内存单元有个编号，这就是“地址”，指针存放的内容就是该内存单元的编号，也就是地址。

1.3.5 指针的类型

指针可以指向不同类型变量的地址 (RAM 地址)，比如 int、char 等，所以在定义指针时必须指定指针要指向的数据类型，特殊的指针类型如指向函数的指针，指向指针的指针，指向多维数组的指针。指针的大小与指针的类型无关，不管是何种类型的指针，其值都固定的且只与体系结构有关，如 char *p; long *q 而使用 sizeof(p) 和 sizeof(q) 时，其值都为 2(C compiler V3)。

1.3.6 指针的操作

操作指针分 3 步：定义指针，初始化指针，使用指针。特别要注意的是在使用指针时必须初始化，如果没有初始化指针就向指针指向的地址写入数值会引起不可预见的错误。指针的操作符有两个：&(取地址操作) 和 *(取内容操作)。

1、操作指向变量的指针：

例子 6：可以改变 a 的值吗？

代码清单 1.8：

```
char a;
void change(char *b)
{
   *b = 'b';
}
void main()
{
   a = 'a';
   change(&a);
}
```

运行结果：a = ‘b’，由于传进来的参数是 a 的地址，改变该地址里面的值就相当于改变实参的值。

2、操作指针的指针：

例子 7：可以改变 a 的值吗？

代码清单 1.9：

```
char *a;
void change(char *b)
{
   b = (char *)0x81;          // 修改指针本身的值
}
void main()
{
   a = (char *)0x80;
   change(a);
}
```

运行结果：a = 0x80，

由于参数的传递是单向的，所以如果要改变 a 的值，则改为如下代码：

代码清单 1.10：

```
char *a;
void change(char **b)
{
   *b = (char *)0x81;         // 修改指针内容
}
void main()
{
   a = (char *)0x80;
   change(&a);
}
```

运行结果：a = 0x81

3、操作数组指针和指针数组：

数组指针是指指向数组的指针，定义形式如：char (*b)[5];

指针数组则是存放数据类型为指针的数组，定义形式如：char* a[5]。

有什么区别呢？在这里 a 是数组的首地址，在数组 a 中存放的是 5 个指标值，而 b 是指标，它指向的是长度为 5 个 char 型的数组的首地址，如果用 sizeof 运算符便知 a 的长度为 10，而 b 的长度为 2。

4、溢出的下标：

代码清单 1.11：

```
unsigned char a8[5] = {0,1,2,3,4};
unsigned char *p = a8;
*(p + 5) = 8;
```

运行结果：如果 a8 在存放在内存为 0x0085 的位置，则 0x008a 地址的值为 8。如果此时 0x008a 存放一个很重要的数据，则影响了整个程序的运行结果。

5、操作常量指针和指针常量：

常量指针表示指针指向的内容是常量，定义方式如：const char *c_p1;

指针常量表示指针本身的值不能变，但指向的内容可以变，定义方式如：char *const p2_c;

代码清单 1.12：

```
void main()
{
   char a[5] = {0, 1, 2, 3, 4}, b[5];
   const char *p = "Hello World!";
   char * const q = a;
   *(p + 1) = 'o';                    // 企图修改字符串常量的内容，报错。
   p = "Hello! World!";               // 可以修改指向常量的指针。
   p = b;                             // 也可以指向非常量地址。
   *(q + 1) = 2;                      // 可以修改指针常量的指向的内容。
   q = b;                             // 企图修改常量指针的值，报错。
}
```

6、指针运算：

指针可以进行加、减、自增、自减等操作，

代码清单 1.13：

```
unsigned int a[5] = {1,2,3,4,5};     // 假设 a = 0x84
unsigned int *p = a;                 //p = 0x84
void main()
{
   p = p + 1;                //p = 0x86, unsigned int 占 2 个字节
   *(++p) = 10;              //p = 0x88
}
```

运算结果：p = 0x88，a[2] = 10

1.3.7 数组名与指针的区别

数组名的本质是该数组在内存中的首地址，与指针的概念神似，但数组名和指针却有绝对的不同之处：

1、数组名是常量，不可修改，相当于指针常量；

2、sizeof 运算结果不同，数组名：占用的空间，指针：固定为 2；

3、数组名不可自增、自减等。

1.4 结构体、联合体与枚举

结构体是数据的集合，与数组不同之处是数组中的每个元素必须是同种类型，而结构体没有这个限制；联合体与结构类似；枚举则是把变量的值限制在一个指定的范围内。

1.4.1 结构体、联合体与枚举的使用

例子 8：寄存器 PA 和 PA5 的定义 (利用结构体和联合体)

代码清单 1.14：

```
#define DEFINE_SFR(sfr_type, sfr, addr) \
static volatile sfr_type sfr __attribute__ ((at(addr)))
typedef unsigned char __sfr_byte;
typedef struct {                       // 定义结构体
   unsigned char __pa0 : 1;
   unsigned char __pa1 : 1;
   unsigned char __pa2 : 1;
   unsigned char __pa3 : 1;
   unsigned char __pa4 : 1;
   unsigned char __pa5 : 1;
   unsigned char __pa6 : 1;
   unsigned char __pa7 : 1;
} __pa_bits;
```

```
typedef union {                          // 定义联合体
   __pa_bits bits;                       // 结构体的使用
   __sfr_byte byte;
} __pa_type;
DEFINE_SFR(__pa_type, __pa, 0x1a);       // 定义 PA 寄存器
#define _pa        __pa.byte             // 联合体成员的使用
#define _pa5       __pa.bits.__pa5       // 结构体成员的使用
```

枚举类型定义时如果没有指定初值，则其值由上一个有赋值的开始，依次加 1。如果全部没有指定初值，则由 0 开始，依次加 1。

代码清单 1.15：

```
enum weekday{sun = 6, mon = 0, tue, wed, thu, fri, sat};
const unsigned char *dayLine;
const unsigned char *printDay[7] = {
   "HI, Monday ! ",
   "I am Tuesday ! ",
   "Today is Wednesday ! ",
   "Today is Thursday ! ",
   "Happy Friday ! ",
   "Today is Saturday ! ",
   "Today is Sunday ! ",
};

const unsigned char* getDay(enum weekday today)
{
   return printDay[today];
}

void main()
{
   enum weekday today = sat;
   dayLine = getDay(today);              //dayLine = "Today is Saturday !"
   unsigned char ch;
   while(*dayLine){
         ch = *dayLine;
         dayLine++;
   }
}
```

1.4.2 结构体与联合体的区别

结构体与联合体的最主要区别是空间的分配，结构体会为各成员分配内存空间，而联合体内各成员共享一个内存空间，以占用内存空间最大的成员为联合体分配内存空间。

如例子 8 中，如果使用 sizeof(__pa_type) 求联合体占用的空间得到结果是 1，如果把 _pa5 置 1，则 __pa_type 内的 byte 成员的第 5 位也会被置 1。

1.5 预处理，宏定义与内联函数的区别

标准 C 语言中预处理有 3 种：宏定义 (#define)、文件包含 (#include)、条件编译 (#if 等)，预处理是在编译之前执行的，如宏展开的时机是在预编译时进行。

1、宏定义：注意使用宏定义时，只是简单的把宏名替换为宏定义的内容，并不做任何运算，如 #define S(r) 3.1415926*r*r，如果使用 S(6+6)，结果就成了 3.1415926*6+6*6+6，而不是最初想要的结果，可以使用 #define S(r) 3.1415926*(r)*(r) 得到想要的结果。

2、文件包含：使用 #include 可以把相关文件包含进来，包含文件时，使用双引号和使用“<>”不同，一般双引号用于用户自定义的文件，“<>”用于库文件，以减少在编译时因搜索路径所花费的时间，注意头文件的重复包含问题。

3、条件编译：选择源代码中的一支来编译，而不用把全部源代码都进行编译，常用在调试和在不同机器上移植代码时使用，有 3 种形式 (else 分支可以没有)：

(1)、#ifdef 条件

程序段 1

#else

程序段 2

#endif

(2)、#ifndef 条件

程序段 1

#else

程序段 2

#endif

(3)、#if 条件

程序段 1

#else

程序段 2

#endif

4、宏定义与内联函数的区别：

(1)、宏定义只做简单的替换，替换时不做任何运算。

(2)、宏定义在预编译时进行，内联函数则是在有函数调用时进行。

(3)、宏定义的参数不能指定数据类型，内联函数则必需要指定。

(4)、编译程序对宏定义本身的内容不做任何检查。如使用 #define error **8，不会报错，而内联函数则不行。

例子 9：预处理综合实例

代码清单 1.16：

```
FUNCTION.H
#ifndef _FUNCTION_H              // 如果去掉 #ifndef，则报重定义出错
#define _FUNCTION_H

typedef unsigned char    u8;
typedef unsigned int     u16;
typedef unsigned long    u32;

#define BOOL       u8
#define TRUE       1
#define FALSE      0

#define LeftShift(val, times)     (val) << (times)
#define Max(num1, num2)    (num1) > (num2) ? (num1) : (num2)
u8 getMax(u8 num1, u8 num2)
{
```

```
   return Max(num1, num2);
}

#endif

KEY.H
#ifndef _KEY_H
#define _KEY_H
#include "FUNCTION.H"
u8 GetKey(u8 num1, u8 num2)
{
   u8 key = getMax(num1, num2);
   return LeftShift(key, 2);
}

#endif
MAIN.C
#include "FUNCTION.H"
#include "KEY.H"
#define DEBUG 1                              // 调试时用

u8 _debugval;

void main()
{
   u8 ch;
   const u8 *Led_String = "YOU";
   while(*Led_String){
          #if DEBUG                          // 如果把 1 改为 0 则不编译该语句
          _debugval = *Led_String;
          #endif
          ch = *Led_String;
          ch = LeftShift(ch, 1);
          GetKey(ch, 0);
          Led_String++;
   }
}
```

如果向宏定义 Max(num1, num2) 中的 num1 传入 ch++，mun2 传入 0，则 (ch++) > (0) ? (ch++) : (0)，此时 ch 的值就与预想的不一样，所以使用宏定义时要多注意。

1.6 流程控制

1.6.1 三种执行流程

C 语言的执行流程有：顺序执行、选择执行、循环执行。

1.6.2 判断语句 if、switch 的使用

if 和 switch 都具有判断的作用，当 if 的条件情况太多时一般使用 switch 替换，switch 的条件的类型只可以是除浮点型之外的基本数据类型和枚举型。case 只能带常量 (不包括 const 申请的常量)，每个 case 执行后要使用 break，否则会继续执行下面的语句，多个 case 可以执行同一些语句，default 是默认的情况，default 可以不加 break。

代码清单 1.17：

```
unsigned char f;
……
switch(f){
case 12:
case 13:
   f += 1;
   break;
case 14:
   f += 2;
   break;
default:
   f += 3;
}
```

1.6.3 循环与循环的嵌套

1、while(表达式) 语句

2、do…while(表达式);

两者的区别：do…while(表达式); 至少会执行循环体内的语句一次。

代码清单 1.18：

`int sum = 0;`
`void main()`
`{`
`  int i = 11;`
`  while(i < 11){`
`  sum += i;`
`  i++;`
`  }`
`}`
	`int sum = 0;`
	`void main()`
	`{`
	`  int i = 11;`
	`  do{`
	`  sum += i;`
	`  i++;`
	`  } while(i < 11);`
	`}`

执行结果：sum = 0	sum = 11

3、循环的嵌套

在循环体中可以再执行循环体。

1.6.4 break 与 continue

break 只能用于循环和 case 语句，continue 只能用于循环语句，两者的区别是：在循环中如果遇到 continue 则执行下一次该循环体的语句，break 则是直接跳出了本层循环体，执行循环体外的语句。

代码清单 1.19：

```
while(1)
{
   int j = 0;
   while(1)
   {
      j++;
      if(j == 5)
      {
         continue;         // 执行本层循环的下一次循环
      }
      if(j == 10)
      {
         break;            // 跳出本层循环
      }
   }
}
```

1.6.5 正确使用 goto

goto 可跳到本函数内任何一个标号处执行语句，一般不建议使用。

例子 10：正确使用 goto 可以优化代码之场景

代码清单 1.20：

```
typedef unsigned char u8
#define BOOL       u8
#define TRUE       1
#define FALSE      0
u8 result;
```

BOOL fun1()	BOOL fun2()	BOOL fun3()
{	{	{
……	……	……
}	}	}

本来版本：

```
void main()
{
       BOOL b_result = FALSE;
       u8 b[3], a[3];
       u8 *p = a;
       b_result = fun1();
       if(!b_result){
           result = 0x55;
           p = b;
       }
       b_result = fun2();
       if(!b_result){
           result = 0x55;
           p = b;
       }
       b_result = fun3();
       if(!b_result){
           result = 0x55;
           p = b;
       }
}
```

goto 版本：

```
void main()
{
       BOOL b_result = FALSE;
       u8 b[3], a[3];
       u8 *p = a;
       b_result = fun1();
       if(!b_result) goto error;
       b_result = fun2();
       if(!b_result) goto error;
       b_result = fun3();
       if(!b_result) goto error;
       retrun;
error:
       result = 0x55;
       p = b;
}
```

例子 11：使用 do…while(0) 消除 goto

do…while(0) 版本：

```
void main()
{
   char b_result = 0;
   u8 b[3], a[3];
   u8 *p = a;
   do{
       b_result = fun1();
       if(!b_result) break;
       b_result = fun2();
       if(!b_result) break;
       b_result = fun3();
       if(!b_result) break;
   }while(0);
   result = 0x55;
   p = b;
}
```

1.7 作用域

无论是变量还是函数，都具有其作用域。全局变量 / 函数在整个工程中使用 extern 后就可以使用，如果在全局变量加 static 则只能在本文件内使用，为了节省 ROM 空间，函数一般不使用 static，局部变量在函数内申请之后的语句都可以使用，如果是在循环，if，switch 内申请的变量，则执行完这些语句之后，下面的语句就不能再使用这些变量，如代码清单 1.19 中的 j 变量，出了外层 while 循环后的语句将不能够再使用。

第二章 C compiler V3 扩展及其限制

2.1 在 HT-IDE3000 中设置 C compiler V3

2.1.1 使用环境

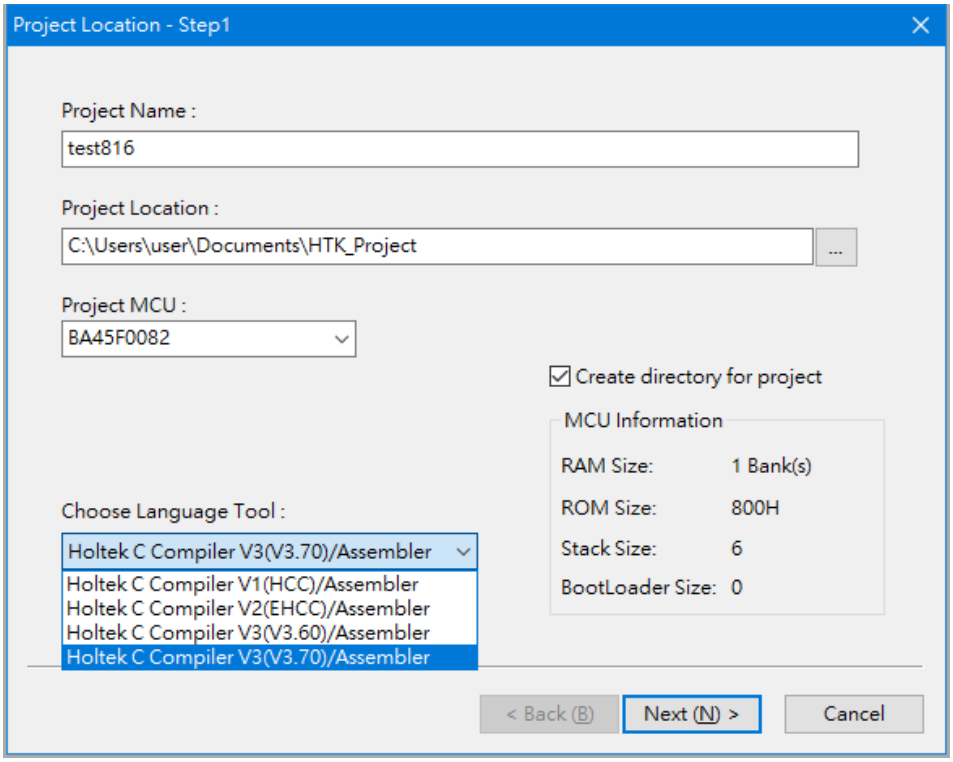
本文于 HT-IDE3000 V8.1.6 的基础上编写。

注：HT-IDE3000 V7.71 开始支持 C compiler V3

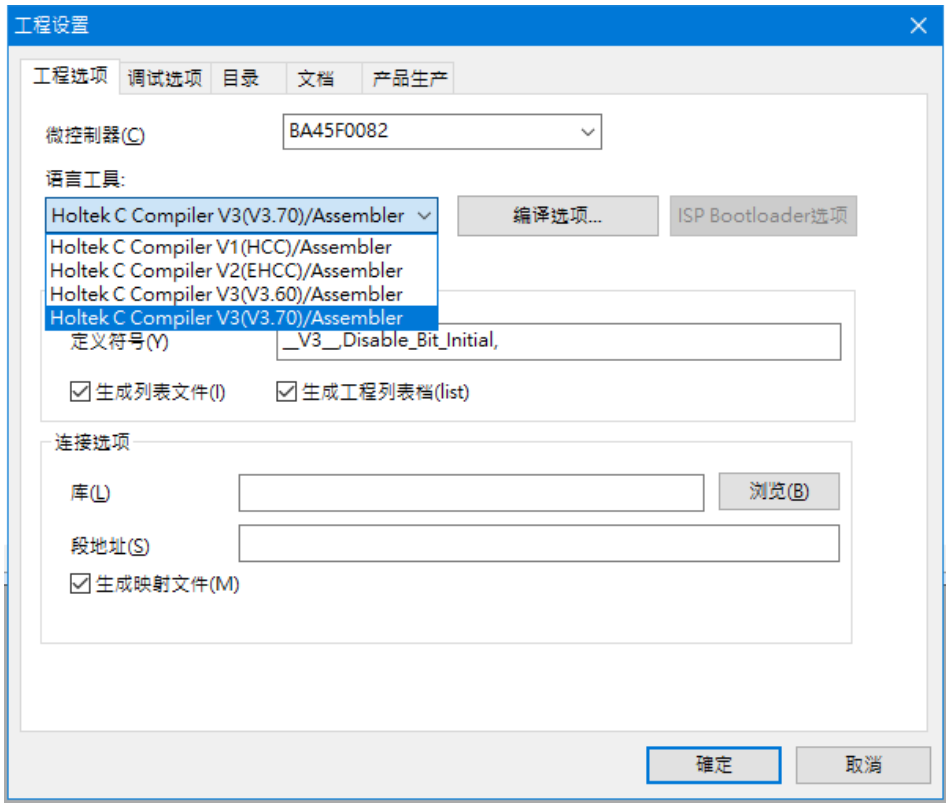
2.1.2 新建项目时，选定 C compiler V3 编译程序

进入 HT-IDE3000 开发环境后，依照下列方法建立一个新的项 (project)：

- 移动鼠标游标到 Project 选单，按左键。
- 移动鼠标游标到 New 命令，按左键。
- 出现如下的窗口，在 Choose Language Tool 之处勾选 Holtek C Compiler V3 (V3.xx)/Assembler。



2.1.3 开启项目后，如何选用 C compiler V3 编译程序



若项目 (project) 已开启之后，可以点选 Option 选单下的 Project Setting 命令，在 Choose Language Tool 中点选 Holtek C Compiler V3(V3.xx)/Assembler 以设定使用 C compiler V3 版编译程序。

2.1.4 项目编译选项设定

1、定义符号

如下图红框设置，IDE 将会给编译程序传参数：-DGCC_COMPILER -DCOUNT=5

相当于宏定义：

```
#define GCC_COMPILER
#define COUNT 5
```

且其有效范围为整个 project，多个宏定义用‘,’隔开。

使用宏的一个例子是条件编译：

E.G.1：

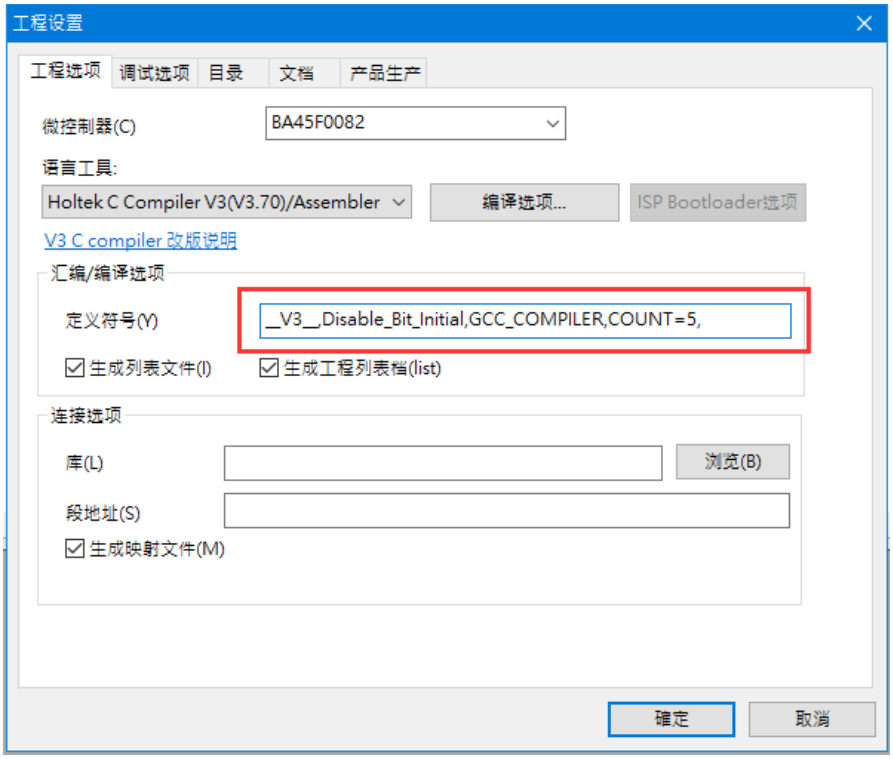
```
#if GCC_COMPILER
typedef unsigned int uint16;
#else
typedef unsigned int uint8;
#endif
```

E.G.2：

```
#if COUNT==5
x = 5
#elif COUNT == 2
```

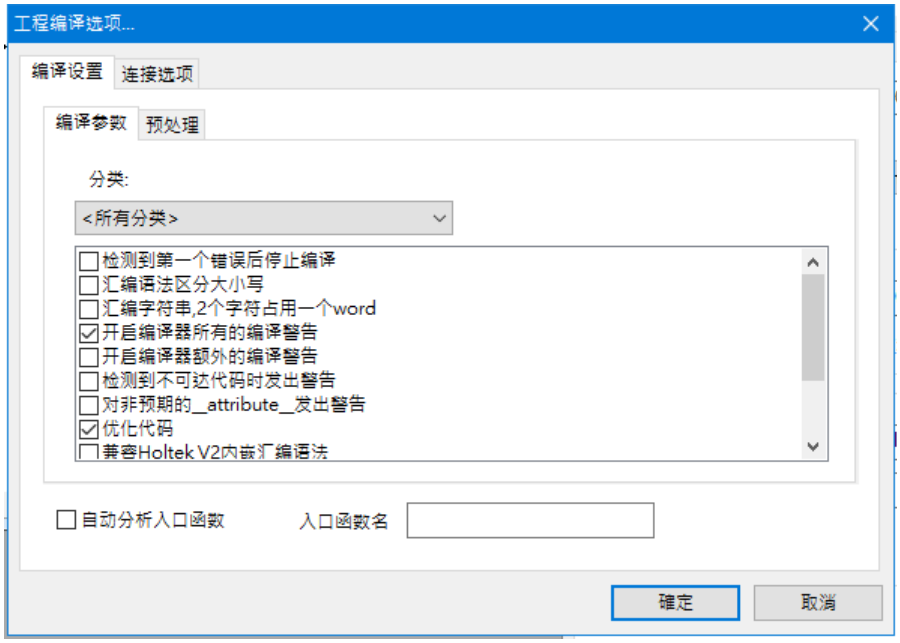
```
x = 6;
#else
x = 7;
#endif
```

注意：使用 C Compiler V3，IDE 会默认为其传宏定义
“__V3__,Disable_Bit_Initial”



2、编译参数

使用 C Compiler V3，可以选择编译选项，IDE 默认会传优化参数 -Os，点击项目设定下的“项目编译选项”，弹出对话框如下：



汇编语法区分大小写：用于混合语言工程，若一个工程既有 c file 又有 asm file 时，可以开启这个选项 (C 语言区分大小写，asm 默认不区分)。

汇编字符串，2 个字符占用一个 word：用于 asm file，当所选 MCU 宽度为 16 bits(如 HT66F50) 时，可以开启这个选项，则字符串中的每两个字符会占用一个 word，否则默认一个字符占用一个 word。若所选 MCU 为 14, 15bits(如 HT46R4AE)，则不可启用这个选项。

开启编译程序额外的编译警告：比如：缺省的函数返回值，无符号数与 0 的比较等等。

优化代码：优化选项。

兼容 Holtek V2 内嵌汇编语法：可以使用 #asm/#endasm 的语法完成内嵌汇编语言功能。

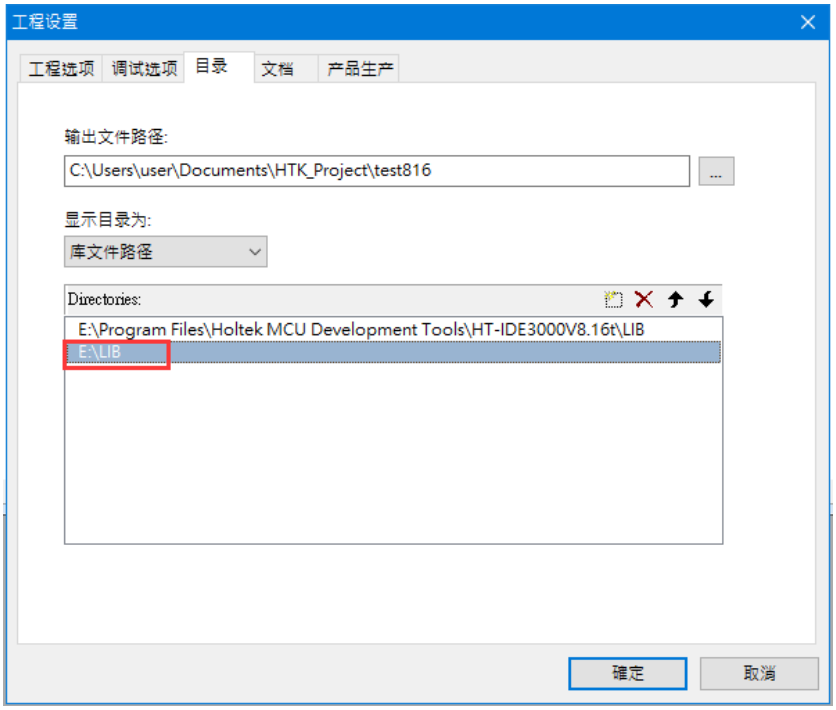
存取 const 变量时使用查表指令进行优化：有 TBHP 寄存器且 PROM 宽度为 16bit 的无扩展指令 MCU 会有此选项，勾选之后 table 的大小减半，若要指定 table 的位置，此选项必须勾选。

配置 enum 类型为 byte：enum 编译器默认为 int，当定义的 enum 数值不大于 127 时，勾选此选项，可以节省空间。

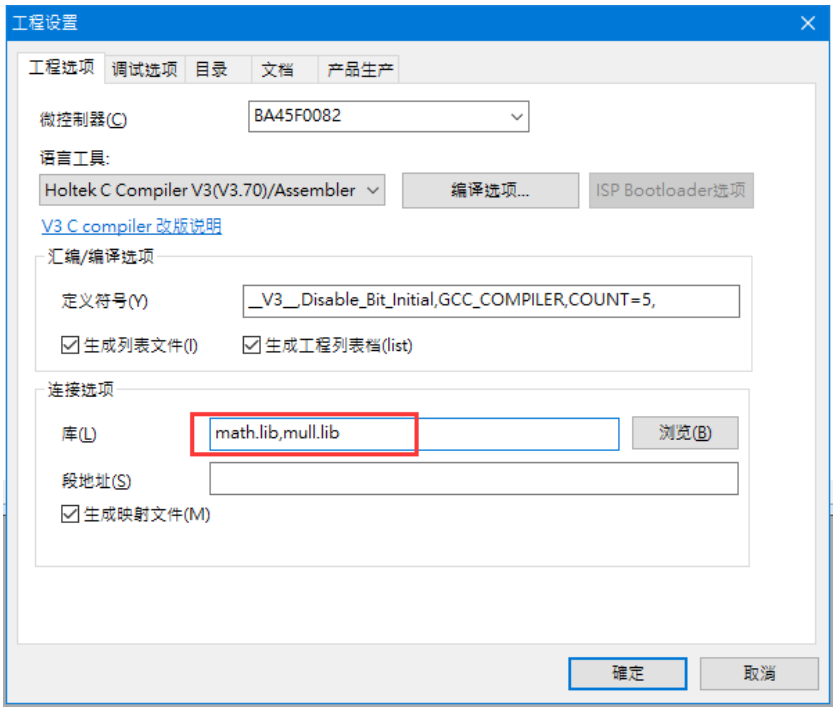
2.1.5 连接选项

2.1.5.1 增加库文件

1、先将库文件路径目录加入目录，如下



2、敲入库文件名，如下 math.lib, mull.lib，用‘,’隔开。



2.1.5.2 设定函数地址

格式：_fun_name=addr,_fun2_name=addr2,…

其中，_fun_name 为 ‘_’ + 函数名称，addr 为所要配置的地址，为 16 进制。多个函数用 ‘,’ 隔开。

注：不建议使用此功能，会影响编译器的优化。

2.1.5.3 生成映射文件

勾选后将生成 .map 文件。

2.1.5.4 连接选项

优化 RAM 空间 (不允许有中断嵌套)：优化 RAM 空间，不允许在一个中断里面进入另一个中断。

删除没有被调用的函数 (针对 C)：若一个函数不被调用，将不为其分配空间，这个选项默认开启。

未指定初值的全局变量 / 静态变量其值默认为 0 (不含 bit 类型变量)：若 global 变量没有初始值，则默认其初始值为 0。在程序开始运行的时候，将其初始化为 0，此操作不包含 bit 类型变量。

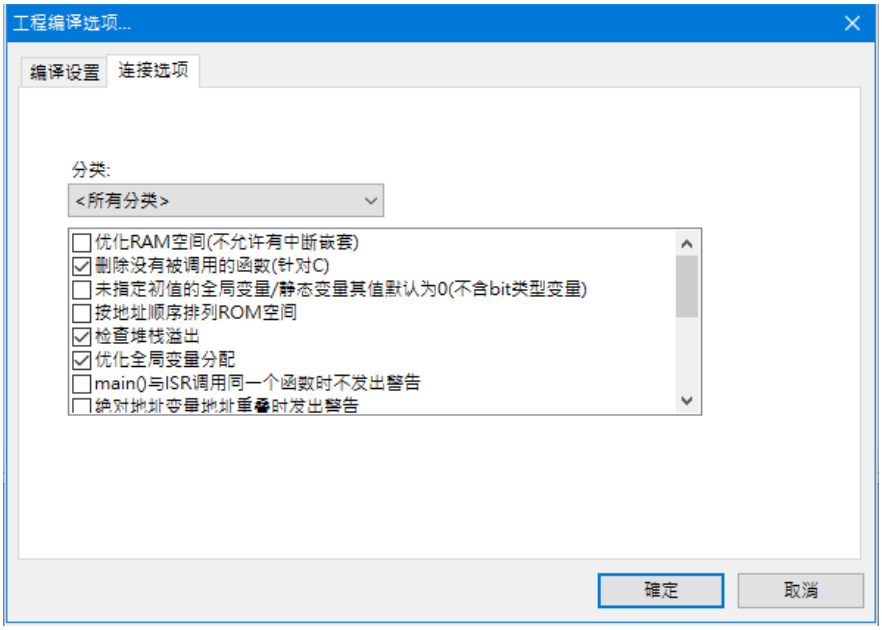
按地址顺序排列 ROM 空间：对于多 ROM bank 的 MCU，为节省 BP 切换指令，编译器不一定会按 ROM bank 顺序分配程序，如果希望按 bank 顺序排列，可以勾选此选项。

检查堆栈溢出：当主程序的堆栈调用层数超过此 MCU 的 stack 数量时，发出警告。

优化全局变量分配：对有扩展指令 MCU，将调用次数多的全局变量优先分配于 RAM bank 0。

main() 与 ISR 调用同一个函数时不发出警告：编译器不允许主程序与中断调用同一个子函数，但如果用户确定此用法没有问题 , 可勾选此选项。

绝对地址变量地址重叠时发出警告：连接器会检查绝对地址变量使用的地址有重叠时发出警告，但如果地址完全重叠，会被认为是同一变量，此时没有警告。



2.2 C compiler V3 扩展语法及功能

2.2.1 中断服务程序

若微控制器的周边装置具有中断功能，程序也需要此中断机能以完成工作时，则必须定义此周边装置的中断服务函数 (Interrupt Service Routine， ISR)，如下的格式：

```
void  __attribute((interrupt(0x0c)))  ISR_tmr0 (void)
{
tick++ ;
}
```

中断服务函数必须遵守下列规定：

→ 返回的数据类型必须是 void

→ 不能有参数

→ 必须使用 __attribute__((interrupt(0x0c))) 设定中断向量值 (interrupt vector)

→ 中断入口会自动保存寄存器 (ACC, BP, STATUS, MP, TBLP, TBHP)，并在中断出口时恢复。

→ 既可以被中断服务程序访问也可以被其他函数访问的全局变量应定义为 volatile。

注：MP/TBLP 只在中断函数有使用到时才会保存。

中断服务函数的使用注意事项：

C Compiler V3 支持中断内部调用函数，但不同中断与 main 之间不能调用同一个函数，会造成 RAM 重叠，对此现象 linker 将侦测出并报 warning (若被调用的函数无宣告及使用任何的 local 变量可忽略此 warning)，如下例子：

```
void fun1(){}
void fun2(){fun1();}
void main()
{
   fun1();
}
void __attribute((interrupt(0x04))) isr1(void)
{
   fun1();
}
void __attribute((interrupt(0x08))) isr2(void)
{
   fun2();
}
```

- main 与 isr1 都调用了 fun1，即为共同调用。
- 虽然 isr2 没有直接调用 fun1，但它调用了 fun2，fun2 又调用 fun1，所以它与 main，isr1 也造成共同调用，调用图可以看 map 文件。
- 同样，各 ISR 之间，也不能调用同一个函数，除非能够保证在中断 1 执行过程中不会进入中断 2。

2.2.2 绝对地址变量 (absolute variable)

将一个变量指定到固定的地址，比如 _bp 在 [04H]，可以写成如下：

```
static volatile unsigned char _bp __attribute__ ((at(0x04)));
```

编译器会将此变量分配到这个地址，但如果这个变量在整个程序中都没有使用到，那么连接器 (linker) 有可能将其它变量分配到这个地址上，编译程序会将之

翻成汇编语言的 EQU 指令，如下：

```
__bp EQU 4h
```

在头文件 ht48c70-1.h 中，有定义它的简写：

```
#define DEFINE_SFR (sfr_type, sfr, addr)\
static volatile sfr_type sfr __attribute__ ((at(addr)))
```

说明：使用 volatile 关键词修饰，以防被优化。

所以如果程序有 include HTxx.h 也可写成：

```
DEFINE_SFR(unsigned char _bp, 0x04);
```

数组、指针、结构体等变量也可以定义成绝对变量，它们的定义方式与一般变量一样，只是多了个地址：

```
static volatile unsigned char arr[10] __attribute__ ((at(0x140)));
// 数组
static unsigned int *volatile p __attribute__ ((at(0x040)));
// 指针
typedef struct
{
  int a;
  int b;
}my_type;
static volatile my_type ab __attribute__((at(0x040))); // 结构体
```

2.2.3 MCU 头文件介绍

HT-IDE3000 有自带 MCU 的头文件，里面主要是一些寄存器及标志位的定义。

在工程中，只需要写 #include “MCU.h”，比如：#include “ht66f60.h”就会自动引入头文件。

头文件的主要内容：

1、绝对地址及中断语法的简写：

```
#define DEFINE_ISR(isr_name, vector)\
void  __attribute((interrupt(vector))) isr_name(void)
```

其中，isr_name 表示中断服务程序的名字，vector 表示中断的地址，比如：

```
DEFINE_ISR(isr_timer,0x0c)
{}
#define DEFINE_SFR(sfr_type, sfr, addr)\
static volatile sfr_type sfr __attribute__ ((at(addr)))
```

其中，static 表示特殊寄存器是静态的，因为每个 C file 都有可能 include MCU 头文件，所以特殊寄存器必须定义成 static，sfr_type 表示它的类型，sfr 为特殊寄存器名称，addr 表示它所在的地址。

比如：DEFINE_SFR(unsigned, _bp, 0x03)

2、特殊寄存器的定义：

```
#define _acc      __acc
```

其中 __acc 已经是用 DEFINE_SFR 定义好的变量，用户可以直接在程序中使用 _acc

3、标志位的定义：

```
#define _c        __status.bits.__c
```

其中 __status 是已经定义好的变量，如果 user 想要自定义标志位，可以参考它的定义方式，如下：

a. 需先定义一个 struct(STATUS 寄存器的所有标志位的集合) 及 union (使用 STATUS 可以整个 byte 操作，也可以单独操作它每一个 bit)：

```
typedef struct {
```

```
        unsigned char __c : 1;
        unsigned char __ac : 1;
        unsigned char __z : 1;
        unsigned char __ov : 1;
        unsigned char __pdf : 1;
        unsigned char __to : 1;
        unsigned char __cz : 1;
        unsigned char __sc : 1;
} __status_bits;

typedef union {
        __status_bits bits;
        unsigned char byte;
} __status_type;
```

b. 指定 STATUS 的地址在 0x0a，如果是一般的 bit 变量则不需要指定地址

```
DEFINE_SFR(__status_type, __status, 0x0a);
```

c. 再将 _pa0 宏定义为其中的一个位域，方便引用。

```
#define _c          __status.bits.__c
```

4、内建函数的定义：

内建函数	作用
GCC_RL(varname)[1]	varname 不带进位左移
GCC_RLC(varname) [1]	varname 带进位左移
GCC_RR(varname) [1]	varname 不带进位右移
GCC_RRC(varname) [1]	varname 带进位右移
GCC_NOP()/_nop()	执行一个空操作 (NOP)
GCC_SWAP(varname) [1]	varname 的前四位与后四位交换
GCC_HALT()/_halt()	执行一个 HALT 指令
GCC_CLRWDT()/_clrwdt()	清零看门狗定时器 (CLRWDT)
GCC_CLRWDT2()/_clrwdt2()	清零看门狗定时器 (CLRWDT2)
GCC_CLRWDT1()/_clrwdt1()	清零看门狗定时器 (CLRWDT1)
GCC_DELAY(n) [2]	延迟 n 个指令周期

注：[1] varname 必须是一个 8-bit 变量
[2] 0 <= n < 263690

5、若 MCU 带 EEPROM，则定义 __EEPROM_DATA(a, b, c, d, e, f, g, h);
可以直接使用此语法对 EEPROM 写值：

```
#define ___mkstr(x)                 #x
#define __EEPROM_DATA(a, b, c, d, e, f, g, h) \
asm("eeprom_data .section 'eeprom'"); \
asm("db\t" ___mkstr(a));        \
asm("db\t" ___mkstr(b));        \
asm("db\t" ___mkstr(c));        \
asm("db\t" ___mkstr(d));        \
asm("db\t" ___mkstr(e));        \
asm("db\t" ___mkstr(f));        \
asm("db\t" ___mkstr(g));        \
asm("db\t" ___mkstr(h));
```

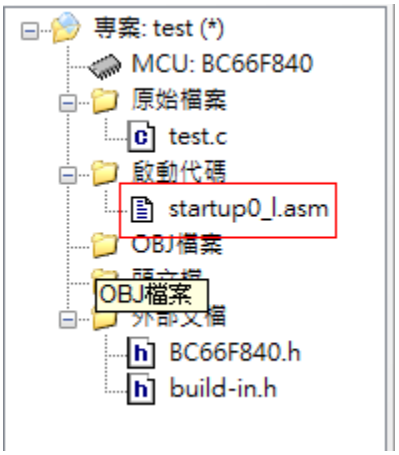
一次可以按顺序写 8 个值，比如：

__EEPROM_DATA(1, 2, 3, 4, 5, 6, 7, 8);

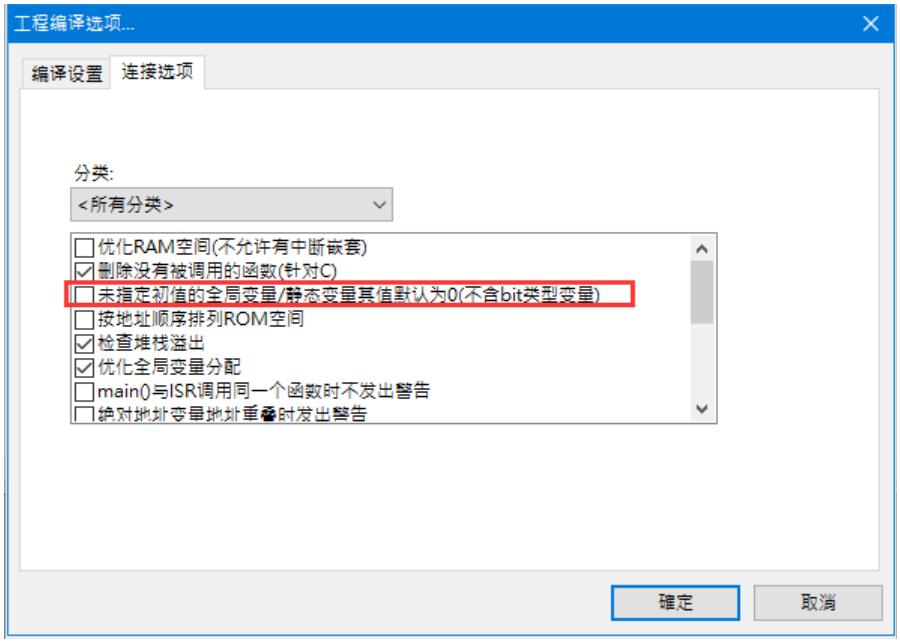
注意，__EEPROM_DATA 不能写在函数体内。

2.2.4 变量初始化

C Compiler V3 支持变量的初始化，并在程序一开始调用一个 startup 的程序来实现，此 startup 程序有两个文文件，startup0 及 startup1：



1、一般建立工程时，未初始化的全局变量默认不初始化为 0，如果有需要再加勾选项，如下，勾选选项即可 (此功能在 IDE7.8 及以上版本实现)



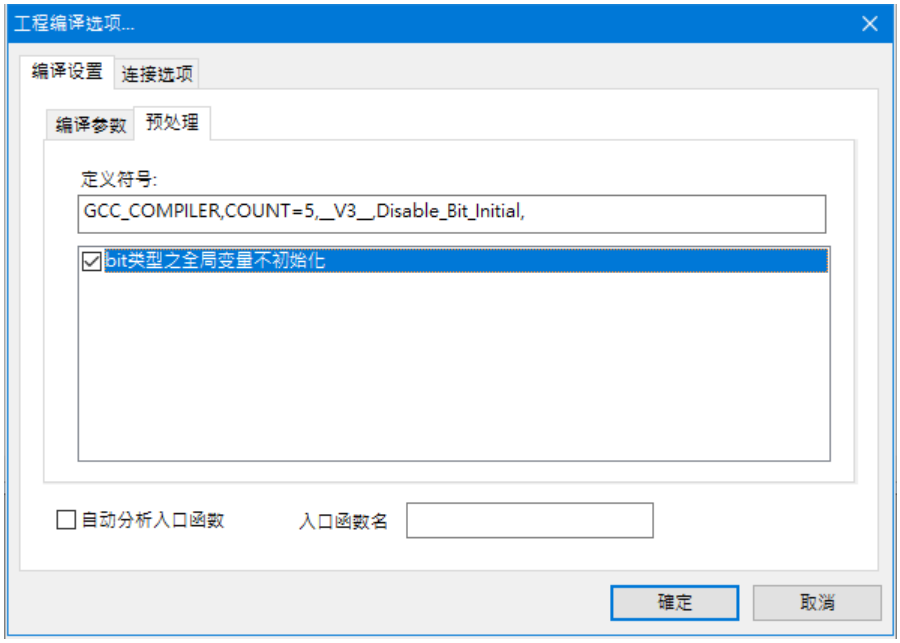
2、Startup 程序的代码是开放的 (此功能在 IDE7.72 以上版本实现)

3、更新 IDE 版本后，需将工程文件夹下的 startup 文件删除，让 IDE3000 重载

4、编译器默认不对全局 / 静态 bit 变量做初始化，如果要修改此设定，可以在此设置 (IDE3000 8.05 及以上版本)：

a. 勾选选项，不管是否有设初始值，编译器都不对 bit 变量初始化。

b. 不勾选选项，编译器对 bit 变量初始化，若无初始值，将默认为 0。



2.2.5 内嵌汇编语言

如果想要让编译后的程序代码更为精简，执行上更有效，可以在 C 程序中加入汇编语言的指令。语法：

1、asm (“opcode [operands]”);

compiler 直接输出指令 opcode [operands]

```
e.g.
extern void FUN() ;
asm("extern _FUN_PAR1:byte");          // 外部函数参数的声明
void main( )
{
  asm("mov a,1");
  asm("mov _FUN_PAR1,a");
  asm("call _FUN");                    // 函数调用
}
```

2、asm (“opcode %0”:“=m”(varname):“m”(varname));

其中 varname 为变量名，为防止被优化，可加 volatile 修饰。

e.g.

```
#include "ht46ru25.h"
char i;
void main()
{
  char c;
  volatile asm("rl %0" : "=m"(_pd): "m"(_pd));       //sfr
  volatile asm("mov a, %0" : "=m"(c): "m"(c));       // 局部变量读值
  volatile asm("mov %0,a" : "=m"(i): "m"(i));        // 全局变量写值
  while(1);
}
```

3、#asm/#endasm

当选择“兼容 Holtek C V2 内嵌汇编语言”编译选项时，可以使用以下语法，输入一整段内嵌汇编

e.g.

```
void main( )
{
  #asm
  MOV A,1
  MOV _FUN_PAR1,A
  CALL _FUN
  #endasm
}
```

注意：

- 每条语句只能写一条指令，需用引号。
- 第二种内嵌汇编格式只能出现在函数内，且变量大小不能超过 1byte。
- 第一种内嵌汇编格式可以出现在函数外，也可以用来定义变量，section 等，其输出内容为引号之间的字符串，编译程序不做处理。

2.2.6 指定函数的地址

C Compiler V3.20 以上版本支持将函数指定地址，语法：

```
unsigned char  __attribute__((at(addr))) fun (char parm){}
```

关键词 __attribute__((at(addr))) 将 function 地址定义在 addr，比如：

```
unsigned char  __attribute__((at(0x373))) foo (char parm){}
```

表示把地址定义在 0x373，指定地址的函数支持参数及返回值。

2.2.7 指定 const 的地址

对 ROM 宽度有 16bit 且带 TBHP 寄存器的 MCU, C Compiler V3.20 以上版本支持将 const 指定地址，语法：

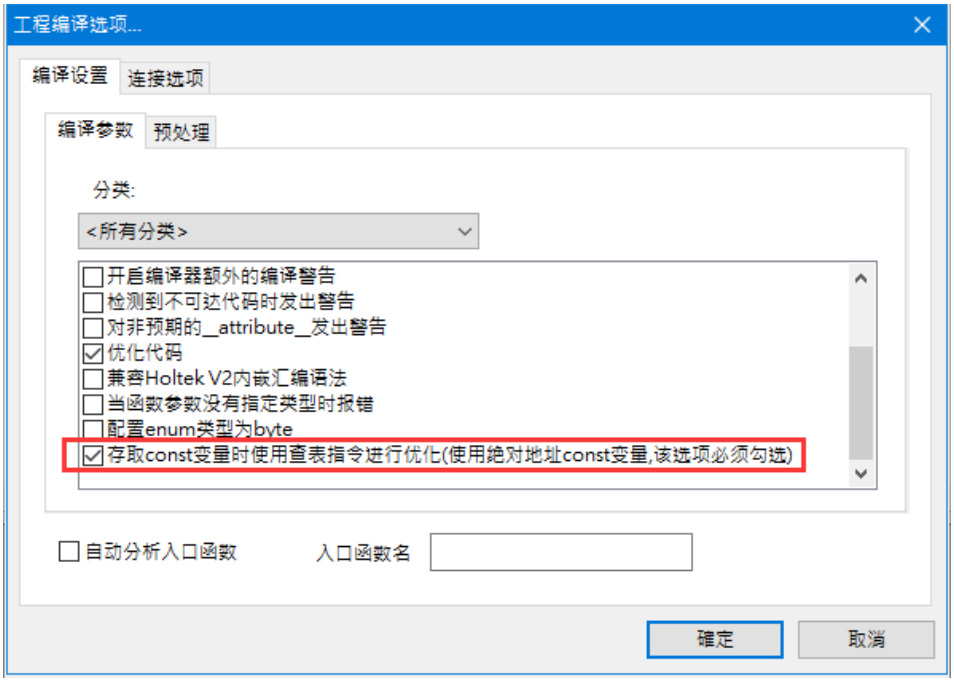
```
const char __attribute__ ((at(addr)))  cvar1 [3]={1,2,3};
```

关键词 __attribute__((at(addr))) 将地址定义在 addr，比如：

```
const int __attribute__ ((at(0x123)))  a[3]={1,2,3};
int b;
int c = 9;
int fun(int *pa,int a)
{
   a+=*pa;
   retrun a;
}
void main()
{
   b=fun(a,c);
}
```

__attribute__ ((at(0x123))) 表示把地址定义在 0x123。

说明：使用该功能需要勾选如下图所示之选项



2.2.8 变量分配

Holtek MCU 有两个 space，ROM 和 RAM。变量的分配规则如下：

1、一般变量会占用 RAM 空间，其初始值存放在 ROM 空间

2、const 变量分配 ROM 空间，但如果使用 volatile 修饰且无指定地址，则占用 RAM 空间

3、如果静态变量在程序中没有被改变值，且传优化参数，则编译程序当其为 const，占用 ROM 空间

2.2.9 __attribute__ 关键词

__attribute__ 是 C Compiler V3 的特有关键词，目前，V3 支持的 __attribute__ 的用法如下：

1、__attribute__((entry))(IDE7.82 以上版本支持)

指定一个函数的属性为入口函数

语法：

```
__attribute__((entry))
void entry_function_name (void){}
```

说明：

a. entry 无参数

b. 返回值类型为 void

c. 参数类型为 void

d. 不限制 main/isr 设定 entry， 但此时 attribute entry 将无效。

范例：

```
__attribute__((entry))
void func (void)
{}
```

2、__attribute__((at(addr)))

指定一个函数 / 变量的地址

语法：

指定函数：

```
__attribute__((at(addr)))
return_type function_name (parameters, …){}
```

指定变量：

```
__attribute__((at(addr)))type variable_name;
```

说明：

a. addr 为指定地址，不可缺失

b. 变量定义了 __attribute__((at(addr)) 属性后需定义成 static

c. const 变量也可定义 __attribute__((at(addr)) 属性，仅限于具扩展指令 MCU 或具有 TBHP, ROM 宽度为 16bit 的 MCU，此时 addr 为变量在 PROM 中的地址。

d. main/isr 也可设定 __attribute__((at(addr)) 属性

范例：

```
__attribute__((at(0x400)))
void func (void)
{}
__attribute__((at(0x180)))
int a;
__attribute__((at(0x100)))
const int array[4]={1,2,3,4};
```

3、__attribute__((interrupt(addr)))

定义一个中断向量入口地址

语法：

```
__attribute__((interrupt(addr)))
void isr_name (void){}
```

说明：

a. addr 是中断入口地址，不可缺省，而且必须为 4 的倍数

范例：

```
__attribute__((interrupt(0x04)))
void isr_timer (void){}
```

一个函数可以定义多种属性：

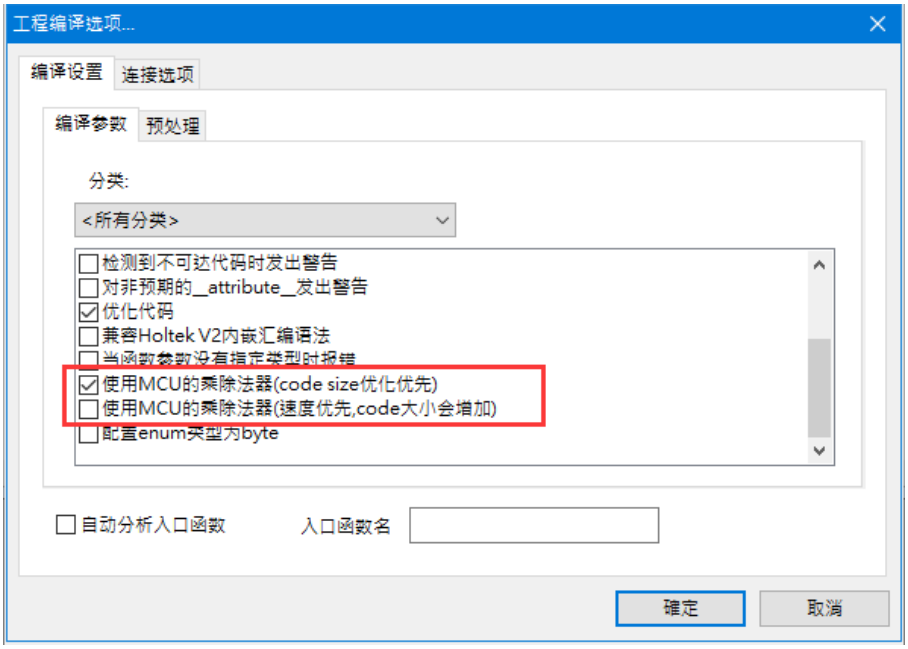
```
a. __attribute__((entry,at(addr)))
b. __attribute__((interrupt(addr),at(addr)))
```

2.2.10 硬件除法器

1、硬件乘除法器 (MDU) 功能适用于具有 MDU 的 MCU，可查看 datasheet, 若有 MDUWR0、MDUWR1、MDU0R0…寄存器，说明此 MCU 有 MDU 功能。

2、IDE3000 V7.90 支持 MDU 功能。

3、对具有 MDU 功能的 MCU，IDE3000 下“选项 → 项目设定 → 编译选项 → 编译参数”会出现以下两个参数：



A. “使用 MCU 的乘除法器 (code size 优化优先)”：表示启用 MDU 功能，为默认勾选选项，需注意，若从旧版 IDE3000(V7.89 及更早) 移植过来的程序，需确定此选项有无勾选。

B. “使用 MCU 的乘除法器 (运算速度优先，code 大小会增加)”：因大部份 MCU 的 MDU 都为 16bit，所以对于 char/unsigned char 类型的乘除法运算，若使用 MDU 功能，使用的 code size 不一定会更少，但是指令周期会更快，所以此选项适用于追求速度优先的用户。

4、大部份 MCU 的硬件乘除法器只有 16bit×16bit、32bit/16bit、16bit/16bit 三种 (HT66FM5440 有 8bit×8bit 及 8bit/8bit)，所以并不是所有的乘除法运算都会使用硬件乘除法器：

乘法	long	U long	int	U int	Char	U char
long	—	—	—	—	—	—
int	—	—	MDU	MDU	MDU	MDU
char	—	—	MDU	MDU	MDU	MDU
U long	—	—	—	—	—	—
U int	—	—	MDU	MDU	MDU	MDU
U char	—	—	MDU	MDU	MDU	MDU

除法	long	U long	int	U int	char	U char
long	—	—	—	—	—	—
int	—	—	—	MDU	—	—
char	—	—	—	MDU	—	—
U long	—	—	—	MDU	—	—
U int	—	—	MDU	MDU	MDU	MDU
U char	—	—	—	MDU	—	MDU

注：a. char 的相关运算要选择参数“速度优先”才能使用 MDU。
b. 除法运算需等式左边类型小于 Long 才能使用 MDU。

c. 除法部份，左列是被除数。
d. U 表示 unsigned。

5、使用硬件乘除法器与乘除法运算库的效益对比：

	乘除法运算库		MDU	
	Size (word)	运行时间 (最坏情况下使用的指令周期)	Size (word)	运行时间
8bit * 8bit	10	106	15	25
8bit / 8bit	25	133	15	25
16bit * 16bit	19	289	19	29
16bit / 16bit	37	330	19	29
32bit * 32bit	31	895	—	—
32bit / 32bit	67	1160	—	—
32bit/16bit	67	1160	19	29

2.2.11 bit 数据类型

1、C Compiler V3.5 以上版本 (HT-IDE3000 V7.93 以上) 开始支持 bit 数据类型。

2、bit 变量占用一个 bit，最低位有效。

3、bit 变量不支援 stuct/union/const/register/ 阵列 / 指针 / 函数参数及返回值。

4、可以指定 bit 变量的地址，语法如下：

```
static volatile __attribute__((at(addr),bitoffset(val))) bit flag1;
```

其中，addr 表示其所在地址单元，val 表示第几位。

比如：

```
static volatile __attribute__((at(0x80),bitoffset(3))) bit flag1;
```

表示 flag1 将占用 [80h].3 的地址。

5、例子：

```
volatile bit flag1;
volatile bit flag2;
static volatile __attribute__((at(0x18),bitoffset(0))) bit pa0;
void main()
{
   while(1)
   {
     if(flag1&flag2)
     pa0 = 1;
   }
}
```

2.3 C compiler V3 的限制

2.3.1 函数指针

C Compiler V3 不支持函数指针，比如如下的程序会报 error：

C:\Users\test\Text1.c:14:10: error: Holtek-gcc does not yet support function pointer.

```
void FileFunc(){}
void EditFunc(){}
void foo()
{
   typedef void (*funcp)(void);
```

```
    funcp pfun= FileFunc;
    pfun();
    pfun = EditFunc;
    pfun();
}
```

2.3.2 递归函数

Holtek MCU 并不支持递归调用，但一种比较特殊的递归调用 (尾递归调用)，编译程序可以做优化将它转化成非递归函数，详见 3.10 节所述。

2.3.3 MP(Memory Pointer) 宽度只有 7bit 的 MCU

部分旧架构的 Holtek MCU 的 MP 只有 7bit，为达到更好的优化效能 C Compiler V3 不支持此类 MCU，具体的 MCU 型号可参考文件 < Holtek C Compiler V3 FAQ >。

2.4 编译程序管理的资源

Holtek MCU 的某些特殊功能寄存器由 C Compiler V3 使用，用户使用时要小心，下表列出这些寄存器，以及它们对编译程序的主要用途，进入中断服务程序时会自动保存所有用到的寄存器。

编译程序用到的特殊寄存器	主要用途
MP/IAR	用于存取 RAM space 的值，搭配 RAM BP 使用
BP	RAB BP 用于存取 RAM space 的值，ROM BP 用于访问 ROM space
STATUS	用于表达式计算
TBLP	用于 table read 或存储 RAM BP
TBHP	用于 table read
TBLH	用于 table read
ACC	存储函数返回值等
PCL	用于 table read

第三章 C compiler V3 的优化功能

3.1 优化内容介绍

C Compiler V3 是一种优化编译程序，其进行优化的主要目的是简化代码，减少代码量。在预设情况下，启用 -Os 将执行所有的优化功能。下表概括了编译程序执行的每项优化功能，包括每个优化功能是否会影响调试。

	是否影响调试	节省 ROM	节省 RAM	节省 Stack
代数转换 (Algebraic Transformations)	√	●		
复制传递 (copy propagation/value propagation)	√	●		
删除执行不到的代码 (Unreachable code Elimination)	√	●		
删除死代码 (Dead-code Elimination)	√	●		
常量折迭 (Constant Folding)		●		
常量传播 (constant propagation)		●		
内联程序 (Inline Procedure)	√			●
强度削减 (Strength Reduction)	√	●		
尾递归调用 (Tail Recursive Call)				●
子表达式删除 (subexpression elimination)		●		
尾部合并 (Tail merging)	√	●		
Bp 优化		●		
Dead section 删除	√	●		

3.2 代数转换 (Algebraic Transformations)

代数转换指的是，compiler 会将表达式的某些运行用更为简洁的相同功能的运算代替，比如：

```
-(-a) → a.
if(!a && !b) → if(!(a || b))
```

这种替换不会影响到调试。

3.3 复制传递 (copy propagation/value propagation)

复制传递是这样一种变换，对于变量 x 和 y，进行 x ← y 赋值后，那么在后面的代码中，只要中间插入的指令没有改变 x 和 y 的值，可用 y 代替 x 。这种优化本身并不能节省指令，但是能实现删除死代码 (请参阅第 2.3.5 节“删除死代码”)。例如下面的 C 源代码：

```
00 char c;
01 void foo (char a)
02 {
03   char b;
04   b = a;
05   c = b;
06 }
```

说明：05 行可以转化为 c=a; 这样 04 行就成为无用代码，可以删去，连同变量 b 也可以删去。

复制传递，会影响到用户的调试，因为，04 行及变量 b 被删去，那么，user 将不能在 04 行设置断点，在变量监视 (watch window) 里面也看不到变量 b。

3.4 删除执行不到的代码 (Unreachable code Elimination)

删除执行不到的代码优化功能删除在正常的程序流程中不会执行的代码。例如下面的 C 源代码：

```
if (1)
{
  x = 5;
}
else
{
  x = 6;
}
```

显然，上面代码片段中的 else 部分绝对不可能执行到。使用此优化后，生成的汇编代码将不包含 else 部分的指令，删除执行不到的代码优化可能对在 C 源代码的某些行设置断点产生影响。

3.5 删除死代码 (Dead-code Elimination)

在函数中计算，但在至函数出口的任何路径上都没有使用过的值视为“死”值。只计算死值的指令视为“死”指令。函数作用域外的值视为使用过 (不是死值)，因为不能确定这种值是否使用过。可以参数第 2.3.3 节“复制传递”中的例子。

删除死代码优化可能会对在 C 源代码的某些行设置断点及变量监视产生影响。

3.6 常量折迭 (Constant Folding)

常量折迭是指 compiler 会对表达式中一些可以内部计算的值直接计算，不为其生成汇编代码：

Example：

```
double a, b;
a = b + 2.0 / 3.0;
```

compiler 将输出如下语句的汇编 →

```
a = b + 0.666667;
```

常量折迭不会影响到调试。

3.7 常量传播 (constant propagation)

一些表达式，虽然里面的变量不是常量，但 compiler 可以通过简单的计算算出它的值，比如：

```
float a=5.0f, b;
b = a + 1.0f;
```

compiler 将输出如下语句的汇编 →

```
float a=5.0f, b;
b = 6.0f;
```

常量传播不会影响到调试。

3.8 内联程序 (Inline Procedure)

当一个子函数比较简单时 (如下条件)，调用它的函数会在调用时直接将它展开，从而减少堆栈的使用：

内联函数的一些规定：

1、函数不能包含复杂的语句，这些复杂的语句包括：switch 语句、for 语句、while 和 do while 语句、goto 语句。

2、函数不能是递归函数。

3、函数体内的函数语句不能超过 30 行。

4、子函数与其调用函数在同一个 C file。

Example：

```
float square (float a)
{
   return a * a;
}
float parabola (float x)
{
   return square(x) + 1.0f;
}
```

内联之后的函数：

```
float parabola (float x)
{
   return x * x + 1.0f;
}
```

内联函数的弊端：

代码量的膨胀，函数每展开一次，它的代码量便增加一次。

如果想要避开内联函数，可以将子函数与母函数使用不同的文件编写。

内联函数的优势：

1、函数展开后，可能有其它的优化可以进行

2、减少堆栈的使用

3、子函数被展开后，若没有被其它函数调用，则会被 Linker 当成 dead section 删去 (2.3.14 节所述)

内联函数会使得被展开的子函数无法设断点。

3.9 强度削减 (Strength Reduction)

强度削减指的是，一些简单循环有可能会被化简为一般的语句，比如：

```
for(i=0; i<10; i++)
{
   j=i;
   k=j+i;
}
```

削减成：

```
i=10;j=9;k=18;
```

强度削减会使得被化简的代码无法设断点。

注意：因为编译程序只关心执行的结果，所以，通常有特殊目的的代码 (比如通过循环来达到 delay 功能) 会因此而失去意义，建议若有此类的代码，可以将循环体用内嵌汇编实现，或者将变量用 volatile 定义。比如：

Example 1：

```
for(i=0; i<10; i++)
{
     asm(”nop”);
  }
```

Example 2：

```
volatile unsigned char count;
for(i=0; i<10; i++)
{
     count++;
  }
```

3.10 尾递归调用 (Tail Recursive Call)

Holtek MCU 并不支援递归调用，但一种比较特殊的递归调用，编译程序可以直接帮助我们做优化，将递归调用直接优化为非递归调用，这一类递归调用在编译程序里面称为尾递归调用 (Tail Recursive Call)。

比如：

```
int primes(int a, int b)
{
    if(a==0)return b==1;
    if(b==0)return a==1;
    return primes(b,a%b);
}
```

像上面的一个代码，只有在代码的最后一行调用函数自身的代码称为尾递归，编译程序将递归优化成一个循环：

```
int primes(int a, int b)
{
 L1:
    if(a==0)return b==1;
    if(b==0)return a==1;
    a = b;
    b = a%b;
    goto L1;
}
```

尾递归调用不会影响 debug。

3.11 子表达式删除 (subexpression elimination)

子表达式删除是指在几个表达式中，若有部份的子表达式是相同的，那么编译程序会先计算这部份，储存在一个临时变量里，之后直接使用此临时变量运算，比如，如下 b*c 为公共的子表达式，可以先计算在 tmp 里，这样可以减少一次 b*c 的计算。

```
int a,b,c,d,g;
a = b * c + g;
d = b * c * d;
→
tmp = b * c;
a = tmp + g;
d = tmp * d;
```

删除子表达式不会影响 User 的 debug。

3.12 尾部合并 (Tail merging)

尾部合并优化将多个相同的指令序列合并为一个指令序列。例如下面的 C 源代码片段：

```
00 if ( user_value )
01 {
02    PORTB = 0x55;
03    user_value=0;
04 }
05 else
06 {
07    PORTB = 0x80;
08    user_value=0;
09 }
```

ln03 与 ln08 是同一段代码，这时编译程序只会编译 ln08 而删去 ln03，这时如

果执行的是 if 分支，则执行完 ln02 时，就会跳到 ln08，有可能会误导 user，以为跑到了 else 分支，但其实不是。

因为两行或更多行源代码可能共享同一汇编代码序列，这使调试器很难确定正在执行哪一行源代码。

3.13 ROM BP 优化

对多个 ROM bank 的 MCU，使用 Jmp 或 call 之前，需设定 ROM BP 来决定是哪一个 bank，如果当前 BP 与 jmp/call 之后的 BP 相同，linker 会删除它多余的 mov BP, a 指令。

若 ROM BP 指令不能删除，则会使用相应的 set/clr BP.5(,6,7) 指令，减少 ACC 的使用。比如：

```
void  fun1()
{
   fun2();
}
```

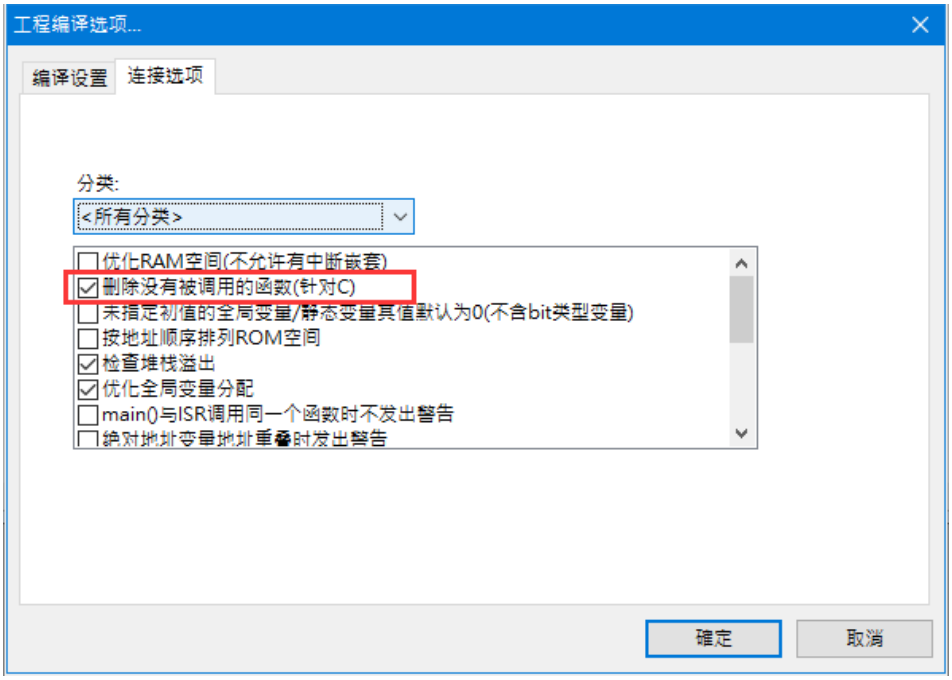
若 fun1 与 fun2 被分配到同一个 bank，则会翻译成 call _fun2

若 fun1 被分配到 bank0，而 fun2 被分配到 bank1，则会翻译成：

```
set BP.5
call _fun2
clr BP.5
```

3.14 Dead section 删除

若一个 function 在整个程序中都不被呼叫，则 linker 不为它分配空间，此功能可以通过如下的选项选择，特别是，当 compiler 将函数调用优化成 inline 展开时，则子函数有可能不被调用而被删除。若此工程含有 ASM 文件，则此功能无效。



第四章 Holtek C V1，V2，V3 及 ANSI C 的差异对比

4.1 数据类型

Data type	Size (bit) V1	Size (bit) V2	Size (bit) V3	Size(bit) ANSI C
bit	1	1	1	N
char	8	8	8	8
signed char	8	8	8	8
unsigned char	8	8	8	8
short	8	16	16	16
unsigned short	8	16	16	16
int	8	16	16	16
unsigned int	8	16	16	16
long	16	32	32	32
unsigned long	16	32	32	32
long long	N	N	32	64
unsigned long long	N	N	32	64
float	N	32	24	32
double	N	32	32	64
long double	N	N	N	128

Holtek C V2、V3 32 位浮点数皆使用 IEEE754 32 位格式

Holtek C V3 24 位浮点数格式：

只有 4~5 位精度 (V3.20 以上版本支持)

符号 sign	指数 e	尾数 m
23	22~15	14~8　7~0

bit 形态不可用于指针 (pointer) 的数据形态，不可定义为 const。

4.2 数组

维数	V1 (最大数组长度)	V2 (最大数组长度)	V3 (最大数组长度)	ANSI C (最大数组长度)
一维数组	256	①	②	不限制
二维数组	N	①	②	不限制
三或三以上的多维数组	N	N	②	不限制
指针数组	N	①	②	不限制
函数数组	N	功能限制	N	不限制
字符串数组	不支持	不支持	②	不限制

注：① 若是 const array，则总长度不超过一个 rombank，若是一般的 array 则不超过一个 rambank。

② 若是 const array，则总长度不超过一个 32K，若是一般的 array 则不超过一个 rambank。

4.3 标识符保留字

保留字	V1	V2	V3	ANSI C
auto	●	●	●	●
break	●	●	●	●
bit	●	●	●	
case	●	●	●	●
char	●	●	●	●
const	●	●	●	●
constant		●		
continue	●	●	●	●
default	●	●	●	●
do	●	●	●	●
double		●	●	●
else	●	●	●	●
enum	●	●	●	●
extern	●	●	●	●
float		●	●	●
for	●	●	●	●
goto	●	●	●	●
if	●	●	●	●
int	●	●	●	●
long	●	●	●	●
register		●	●	●
return	●	●	●	●
short	●	●	●	●
signed	●	●	●	●
sizeof	●	●	●	●
static	●	●	●	●
struct	●	●	●	●
switch	●	●	●	●
typedef	●	●	●	●
union	●	●	●	●
unsigned	●	●	●	●
void	●	●	●	●
volatile	●	●	●	●
while	●	●	●	●
vector	●	●		
__attribute__			● ①	
at			● ②	
interrupt			● ③	
entry			● ④	

注：①与②用于定义绝对地址变量，比如：
unsigned char sfr __attribute__ ((at(0x40))); 表示将变量 sfr 定义于 0x40 地址
①与③用于定义中断向量，比如：
void __attribute((interrupt(0x04))) isr_name(void) {…} 表示在 0x04 地址定义中断 isr_name
④用于指定入口函数，详见 2.2.9 节

4.4 运算符

运算符	V1	V2	V3	ANSI C
算术运算符 (+,-,*,/,%)	●	●	●	●
关系运算符 (>,<,==,>=,<=,!=)	●	●	●	●
逻辑运算符 (!,&&,||)	●	●	●	●
位运算符 (<<,>>,~,|,^,&)	●	●	●	●
赋值运算符 (=,+=,-=,*=,/=,%=,>>=,<<=,&=,^=,|=)	●	●	●	●
条件运算符 (? :)	●	●	●	●
逗号运算符 (,)	●	●	●	●
指针运算符 (* 和 &)	●	●	●	●
求字节数运算符 (sizeof)	●	●	●	●
强制类型转换运算符 (类型))	●	●	●	●
分量运算符 (. ->)	●	●	●	●
下标运算符 ([ ])	●	●	●	●
函数调用运算符 (( ))	●	●	●	●
自增运算符 (++)	●	●	●	●
自减运算符 (--)	●	●	●	●
负号运算符 (-)	●	●	●	●
正号运算符 (+)	●	●	●	●
指定 RAM 变量地址运算符 (@)	●	●		

4.5 前置处理指令

前置处理指令	V1	V2	V3	ANSI C
#asm	Y	Y	N ②	N
#define	Y	Y	Y	Y
#elif	Y	Y	Y	Y
#else	Y	Y	Y	Y
#endif	Y	Y	Y	Y
#error ①	Y	Y	N	N
#if	Y	Y	Y	Y
#ifdef	Y	Y	Y	Y
#ifndef	Y	Y	Y	Y
#include	Y	Y	Y	Y
#pragma	Y	Y	N	N
#undef	Y	Y	Y	Y

注：①产出错误信息：#error size too big
② HOLTEK C V3 的内嵌汇编使用 asm(“”) 格式，详见 2.4。

4.6 预处理指令 #pragma

格式：

```
#pragma  keyword  [option]
```

某些 keyword 会有 options。

Keyword	V1	V2	V3	ANSI C
bp_free		●		
bp_nofree		●		
function		●		
nobp		●		
nolocal		●		
nomp0		●		
nomp1		●		
rambank0 norambank	●	●		
rombank0 norombank		●		
rombank		●		
vector	●	●		
novectornest		●		
inline			● ②	

注：② HOLTEK C V3 支援 inline 函数，其格式与标准 C 同。

4.7 const 变量

Const 变量功能	V1	V2	V3	ANSI C
适用的数据类型	除 bit	除 bit	除 bit	any
直接被其它文件使用	N	N	Y(引用时，在 const 前加修饰词 extern)	Y(引用时，在 const 前加修饰词 extern)
必须宣告为全局型	Y	N	N	N
宣告时要设定初始值	Y	Y	Y	Y
数组常量要指定数组的大小	Y	Y	Y	Y
取址操作数	N	N	Y	Y

4.8 预定义的头文件

预定义的头文件	V1	V2	V3	ANSI C
HTxxxxxx.h	Y	Y	Y	N
assert.h	N	N	N	Y
ctype.h	N	Y	Y	Y
errno.h	N	N	N	Y
float.h	N	N	N	Y
limits.h	N	N	N	Y
locale.h	N	N	N	Y
math.h	N	Y	Y	Y
setjmp.h	N	N	N	Y
signal.h	N	N	N	Y
stdarg.h	N	N	N	Y
stddef.h	N	N	N	Y
stdio.h	N	N	N	Y
stdlib.h	N	Y	Y	Y
string.h	N	Y	Y	Y
time.h	N	Y	Y	Y

4.9 main 函数

main 函数的规定	V1	V2	V3	ANSI C
个数 (个)	1	1	1	1
返回数据类型	void	void	void	int
参数 (个)	无	无	无	2 (一个指针数组)

4.10 中断函数

中断函数的规定	V1	V2	V3	ANSI C
设定中断向量值	Y	Y	Y	没有中断函数
个数 (个)	可多个	可多个	可多个	
返回数据类型	void	void	void	
参数	无	无	无	
重复进入中断	N	Y ①	Y ①	
在程序中调用中断	N	N	N	
中断调用汇编函数	Y	Y	Y	
中断调用 c 函数	N	Y ②	Y ③	

注：①虽然不同的中断事件可以重叠发生，但是同一个中断事件不可以重叠发生，必须等候前一个发生被处理完成后，才能认可下一个中断事件。针对不具有中断可重叠 (nested) 发生的微控制器，则在中断服务函数内不可开启中断功能。

②必须将被调用的函数定义成 #pragma nolocal。否则会造成 RAM 空间重叠使用，一般不推荐使用。

③中断调用的函数不能与 main 函数调用同一个函数，否则会造成 RAM 空间重叠使用，不同的中断也不能调用同一个函数，比如：

isr1 → fun1 → fun3

main → fun2 → fun1
isr2 → fun3
则 isr1 与 main 共同调用了 fun1，isr1 与 isr2 共同调用了 fun3

4.11 内建函数

函数	V1	V2	V3	ANSI C
_clrwdt()	Y	Y	GCC_CLRWDT()	N
_clrwdt1()	Y	Y	GCC_CLRWDT1()	N
_clrwdt2()	Y	Y	GCC_CLRWDT2()	N
_halt()	Y	Y	GCC_HALT()	N
_nop()	Y	Y	GCC_NOP()	N
_rr(int8 *)	Y(int *)	Y(char *)	GCC_RR(int 8)	N
_rrc(int8 *)	Y(int *)	Y(char *)	GCC_RRC(int 8)	N
_lrr(int16 *)	Y(long *)	Y(int *)	N	N
_lrrc(int16 *)	Y (long*)	Y(int *)	N	N
_rl(int8 *)	Y(int *)	Y(char *)	GCC_RL(int 8)	N
_rlc(int8 *)	Y(int *)	Y(char *)	GCC_RLC(int 8)	N
_lrl(int16 *)	Y(long *)	Y(int *)	N	N
_lrlc(int16 *)	Y(long *)	Y(int *)	N	N
_swap(int8 *)	Y(int *)	Y(char *)	GCC_SWAP(int 8)	N
_delay (unsigned long tick)	Y(tick<=65535)	Y(tick<= 263690)	GCC_DELAY(tick)	N

4.12 其它的功能

功能	V1	V2	V3	ANSI C
内嵌式汇编语言	Y	Y	Y ②	N
静态变量	不支持静态变量和静态函数	不支持静态变量和静态函数	Y	Y
常量	支持二进制常量	支持二进制常量	支持二进制常量	不支持二进制常量
结构体和共用体	bit field 置放于 8 位的单位内，不会横跨两个 8 位的单位，且不能定义超过 9 位的 bit field	bit field 置放于 8 位的单位内，不会横跨两个 8 位的单位，且不能定义超过 9 位的 bit field	最大可定义 32 位的 bit field	最大可定义 32 位的 bit field
函数	不支持递归函数	不支持递归函数	不支持递归函数	支持递归函数
指针	不能用于常量与位变量，不支持函数指针	不能用于常量与位变量，若指向函数，则必须是全局的，且所指函数不能带有参数	不支持函数指针	Y

功能	V1	V2	V3	ANSI C
初始值	全局变量宣告时不可以同时定义初始值，但是 const 常量在宣告时一定义要设定初始值	全局变量宣告时不可以同时定义初始值，但是 const 常量在宣告时一定义要设定初始值	Y	Y
堆栈	层数有限①	层数有限①	层数有限①	层数不受限制

注：①每个 MCU 的层数有限，调用函数时，要考虑占用的堆栈层数，一些运算符或函数在调用时所占用的堆栈层数如下不：

运算符 / 函数	堆栈层数	运算符 / 函数	堆栈层数
main()	0	_rl(int */ char *);	0
_clrwdt()	0	_rlc(int *);	0
_clrwdt1()	0	_lrl(long */ int *);	0
_clrwdt2()	0	_lrlc(long *);	0
_halt()	0	_delay(unsigned long)	1
_nop()	0	*	1
_rr(int */ char *);	0	/	1
_rrc(int *);	0	%	1
_lrr(long */ int *);	0	array/pointer	1
_lrrc(long *);	0	整型与浮点型转换	1

② HOLTEK C V3 内嵌汇编格式，详见 2.2.5

第五章 命令行模式

本章主要是使用 C compiler V3 的命令行模式并引导程序员使用命令行模式编译一个源文件。

主要包含如下内容：

- 进入命令行环境
- 使用命令模式生成目标文件的过程
- 命令行参数

5.1 设置环境变量

在使用命令行环境之前，先设置环境变量。

在命令行下使用 set 命令追加 IDE 安装路径下的 bin 目录，例如，

set PATH=%PATH%;XXX/bin

其中 XXX/bin 是 IDE 安装路径下的 bin 目录，这样就可以在本次的 CMD 窗口中使用 bin 目录下的各 IDE 工具

5.2 使用命令模式编译源文件的过程

在设置好了环境变量后，就可以开始编写代码，这里以例子 5 为例，把例子 5 中的 file1.c 和 file2.c 编译成对应的汇编文件。

命令：hgcc32 [options] cfile –o asmfile

编译 file1.c：hgcc32 -g -Os file1.c -o file1.asm

编译 file2.c：hgcc32 -g -Os file1.c -o file2.asm

由上述的指令即可产出编译后的 asm 文件。

5.3 命令行参数

参数	含义
-g	产生 debug 信息
-O0/-O1/-O2/-O3/-Os	优化参数
-D<macro>[=<val>]	宏定义
-I<path>	设定搜寻头文文件的目录 path
-msingle-ram-bank	单个 RAM
-mmulti-ram-bank	多个 RAM(default)
-msingle-rom-bank	单个 ROM
-mmulti-rom-bank	多个 ROM(default)
-fno-builtin	不使用 gcc 内建函数
-mno-tbhp	没有 TBHP
-mtbhp=addr	指定 TBHP 地址 addr(default 为 09H)
-mlong-instruction	扩展指令 MCU

五种优化参数：-O0、-O1、-O2、-O3 和 -Os。编译时只能设置其中的一种来编译。

各个优化等级：

-O0：这个等级 (字母“O”后跟个 0) 关闭所有优化选项，也就是没有设置 -O 等级时的默认等级。这样就不会优化代码。

-O1：这是最基本的优化等级。编译程序会在不花费太多时间的同时试图生成更

快更小的代码。这些优化是非常基础的，但一般这些任务肯定能顺利完成。

-O2：-O1 的进阶优化。这是较优化的等级，-O2 会比 -O1 启用多一些标记。设置了 -O2 后，编译程序会试图提高代码性能而不会增大体积和大量占用编译时间。

-O3：这是最高的优化等级。用这个选项会延长编译代码的时间，而且有可能会因为这个优化的程度而导致产出的代码会偏离原来的代码，一般不使用该优化等级。

-Os：这个等级用来优化代码尺寸。其中启用了 -O2 中不会增加存储空间占用的代码生成选项。这对于存储空间较小的机器非常有用。

更多参数介绍请参考 GCC 使用手册。

第六章 多文件编程

在一个工程中难免要使用多个源文件，把类似的函数和定义编写到同一个文件中，以方便管理，本章将概要描述多文件编程的相关内容。

主要包含如下内容：

- 头文件
- 共享的变量
- 调用其他源文件中的函数
- 使用库

6.1 头文件

头文件用来宣告变量与函数，定义宏、指定地址变量及类型。不可以定义一般变量及函数。

另外，为了避免头文件的内容重复定义，在创建头文件后需要加上。

```
#ifndef _XXX_H
#define _XXX_H
……
#endif
```

6.2 共享的变量

如果编程时多个源文件需要共同访部问同一个变量，一般要把该变量定义为全局变量，并且不能加 static，这里用 extern 的方式声明该变量是外部文件的变量，这样就可以使用该变量了；在头文件中也一样这么使用。extern 可以出现在函数体内，也可以出现在函数体之外。

6.3 调用其他源文件的函数

使用头文件的方式和使用 extern 的方式可以引用外部的函数，使用时要注意调用外部的函数时，这些外部的函数可能会修改其所在源文件中的全局变量。

例子 12：使用外部函数修改外部变量

代码清单 5.1：

```
FUNCTION.H
#ifndef _FUNCTION_H
#define _FUNCTION_H

typedef unsigned char    u8;
typedef unsigned int     u16;
typedef unsigned long    u32;

#define BOOL       u8
#define TRUE       1
#define FALSE      0

u8 getMax(u8 num1, u8 num2);

#endif
FUN.C
#include "FUNCTION.H"
u8 g_var;
u8 getMax(u8 num1, u8 num2)
{
   if(num2 == 0){
```

```
        g_var = 0x55;
    }else if(num1 == 0){
        g_var = 0xaa;
    }
    return num1 > num2 ? num1 : num2;
}

MAIN.C
u8 sk = getMax(28, 0);
void main()
{
    while(1){
        GCC_NOP();
    }
}
```

运算结果：sk = 28，g_var = 0x55

6.4 使用函数库

6.4.1 制作函数库

请参考《HT-IDE3000 使用手册》第九章 函数库总管

6.4.2 制作函数库注意事项

不同的 MCU 型号共用函数库应注意：

1. 指令类型相同，有扩展指令的 MCU 不可与无扩展指令 MCU 共用
2. 单一 ROM/RAM bank 的 MCU 不可与多 ROM/RAM bank 的 MCU 共用
3. 无 TBHP 暂存器与有 TBHP 暂存器的 MCU 不共用
4. ROM 宽度为 16bits 的 MCU 不与 14/15 bits 共用
5. 工程所选择的参数与制作函数库时的参数相同

6.4.3 引用函数库

请参考 2.1.5

第七章 混合语言编程

为了提高程序的效率和 ROM 的利用率，使得合理的利用混合语言编程十分有必要，本章将讲述如何使用混合语言来编写程序。

主要包含如下内容：

- 数据存储格式
- C 语言调用汇编语言函数
- 汇编语言调用 C 语言函数

7.1 数据存储格式

采用小端数存储模式 (Little Endian)，即一个数据的低字节存放在低地址处，而高字节存放在高地址处，例如：

```
static long ldata __attribute__ ((at(0x180)));
ldata = 0xAABBCCDD;
```

数据在存储器中的存放结果如下：

地址	0x180	0x181	0x182	0x183
内容	0xDD	0xCC	0xBB	0xAA

7.2 变量，函数的命名规则

1、Holtek C V3 编译程序在编译全局变量 (global variable) 及函数时，会在原名之前附加一个底线字符 (underscore)，例如：

全局变量 count 编译后改为 _count

函数 GetTotalSize 编译后改为 _GetTotalSize

局部变量，静态变量，函数参数等命名则比较不规则，可以参考其编译出的汇编文件中的 debug 信息，但要注意，每次编译后的名字有可能不同。

2、汇编器 (Assembler) 在编译汇编程序后，会将名字转换成大写字母 (upper case)，例如：

变量 count 编译后改为 COUNT

函数 GetTotalSize 编译后改为 GETTOTALSIZE

7.3 C 语言调用汇编语言函数

汇编语言程序的函数定义规则：

1、将底线字母 (underscore) 加入函数名字之前，并且宣告为公用变量 (public variable)。

2、如果函数具有参数，将其宣告为公用变量。

3、函数中若有局部变量，使用 local 定义。

4、使用 proc/endp 定义函数。

C 程序的调用规则：

1、以大写字母定义并宣告要调用的函数名字。

2、若被调用的函数具有参数，则要外部宣告参数。

3、调用此函数。

例子 13：汇编语言减法函数

代码清单 6.2：

```
CODE.ASM
```

```
public _opera
public _opera_var1
public _opera_var2                ;; 将函数及参数定义成 public，方便外部调用

_opera .section page 'code'
_opera proc                       ;; 使用 proc/endp 定义函数
   local _opera_var1      db ?    ;; 参数定义
   local _opera_var2      db ?
   local _result_local    db ?    ;; 局部变量定义
   mov a, _opera_var1
   sub a, _opera_var2
   mov _result_local, a
   ret
_opera endp
```

```
MAIN.C
extern unsigned char OPERA();    // 以大写字母定义函数名字
asm("extern _OPERA_VAR1  : byte");
asm("extern _OPERA_VAR2  : byte");

void main()
{
   volatile unsigned char result;
   asm(“mov a, 20h”);
   asm(“mov _OPERA_VAR1, a”);

   asm(“mov a, 10h”);
   asm(“mov _OPERA_VAR2, a”);

   result = OPERA();              // 函数调用
   while(1){
          asm("nop");
   }
}
```

执行结果：result = 0x10

7.4 汇编语言调用 C 语言函数

C 程序的调用规则：以大写字母定义并宣告被调用的函数名字。

汇编程序的函数定义规则：

1、将函数名字宣告为外部名字，需在函数名前加底线“_”。

2、调用函数用 proc/endp 声明。

3、如果函数具有参数，则将所对应的变量宣告为外部变量，参数的汇编名字，可以参考 C 函数所编译出的汇编文件，注意，每次编译的结果可能不同，所以尽量不要带参数。

4、调用 C 函数后，读出返回值，若返回值为 1byte，则存于 ACC，若为 2byte，则低字节存于 ra，高字节存于 rb，若为四 byte，由低字节到高字节分别存于 ra、rb、rc、rd，其中 ra~rd 都已定义好，只需宣告就可以使用。

代码清单 6.3：

```
FUN.C
int DISPLAY(char row, int col)   // 定义函数，函数名必须大写
{
  int retval;
  retval=(int)(row << 1) + col;
```

```
  return retval;
}

CODE.ASM
                                    ;;code.asm 调用函数 _DISPLAY
extern _DISPLAY : near              ;; 函数宣告为外部名字
extern _DISPLAY_2 : byte            ;; 宣告参数变量名
extern ra : byte                    ;; 宣告返回值
extern rb : byte
CODE .section 'code'
_code proc
local _code_loc db 2 dup(?)         ;; 局部变量定义
MOV A, 10h
MOV _DISPLAY_2, A                   ;; 存值到第二个参数 col 的低字节
CLR _DISPLAY_2[1]                   ;; 第二个参数的高字节设为 0
MOV A, 20h
MOV _DISPLAY_2[2], A                ;; 存值到第一个参数 row
CALL _DISPLAY                       ;; 调用 C 函数 _DISPLAY
MOV A,ra
MOV _code_loc,A
MOV A,rb
MOV _code_loc[1],A                  ;; 将返回值从 ra, rb… 读出，存入局部变
                                    ;; 量 _code_loc, 低字节 ra, 高字节 rb
RET
_code endp
```

运算结果：变量 code_loc = 0x0050

注意如果链接时出现如图 6_3_1 的错误所示：

```
Error(L2001) : Unresolved external symbol 'RA'
Error(L2001) : Unresolved external symbol 'RB'
```

图 6_3_1

则需要在编译参数中勾选“Case sensitive for assembly”选项。

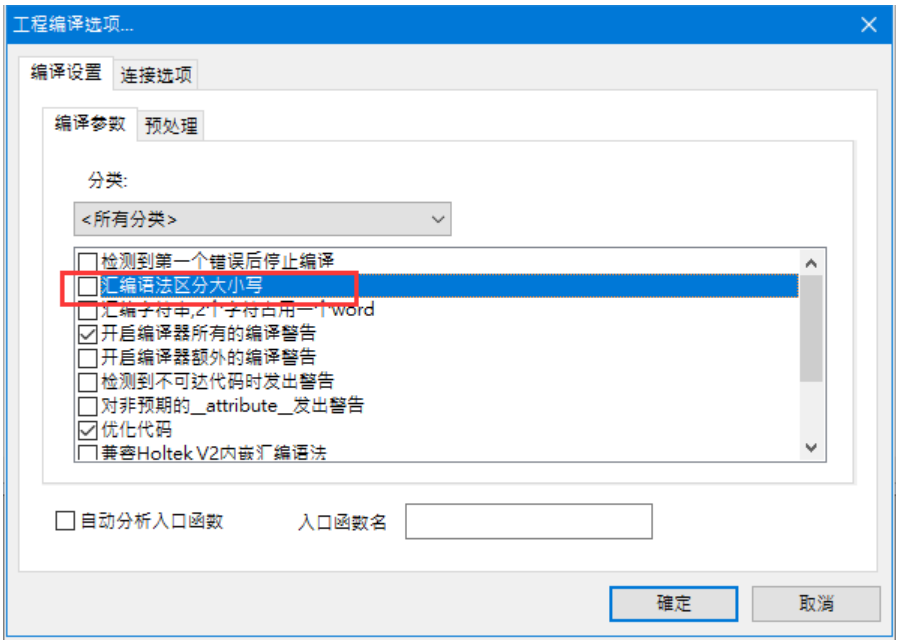


图 6_3_2

第八章 常见错误的解决方式

8.1 内部错误 (Internal Error)

若 error 信息有 internal compiler error 字样，则为 compiler 内部错误，请与 Holtek 公司反馈，比如：

```
ting\ROM_Bank_Setting.c: In function 'main':
ting\ROM_Bank_Setting.c:72:22: internal compiler error: in emit_library_call_value_1, at calls.c:3929
```

8.2 RAM bank0 溢出

对于无扩展指令架构的 MCU，C Compiler 会默认把变量配置到 RAM bank0 (有扩展指令的 MCU 可以自动配置到任意 bank)，当 bank0 满了之后，会报 RAM bank 0 overflow，如下：

```
Linking...
Error(L1038) : RAM (bank 0) overflow,memory allocation fails for section '_fun1'
Total 1 error(s), Total 0 Warning(s)
'ROM_Bank_Setting' - Total 1 error(s), 0 warning(s)
```

出现此信息后，做法如下：

- 检查数据类型是否误用 (特别是从 V1 C Compiler 移植过来的程序)
- 若为 multi RAM bank MCU，可手动将全局变量调到其它 bank，参考 2.2.2 节

8.3 ROM/RAM 空间溢出

当 ROM 或 RAM 空间不够时，会出现这个信息：

```
Error(L1038) : ROM (bank *) overflow,memory allocation fails for sec
Total 1 error(s), Total 43 Warning(s)
'eWriterProLCD_000013' - Total 1 error(s), 45 warning(s)
```

出现此信息后，做法如下：

- 检查是否打开优化参数 -Os，参考 2.1.4 节
- 查看 map 文件，了解 RAM/ROM 分配情况，删减不必要的程序。

8.4 变量重叠警告

绝对地址变量位置重叠，信息如下：

```
Linking...
Warning(L3010) : (Absolute Adress:80H,length:8) is overlay with(Adress:80H,length:8)
Warning(L3010) : (Absolute Adress:88H,length:6) is overlay with(Adress:88H,length:6)
Warning(L3010) : (Absolute Adress:8eH,length:13) is overlay with(Adress:8eH,length:13)
```

出现此 warning 有两种可能情况：

- 同一个绝对地址变量在不同文件定义多次，比如，在 a.h 中定义变量 var：

  ```
  static volatile unsigned char var __attribute__ ((at(0x180)));
  ```

  在 t1.c 与 t2.c 同时 include a.h 就会报这个 warning，对于这种情况，此 waring 信息可以忽略，也可以通过选项设定避免此 warning，参考 2.1.5 节
- 不同变量定义的地址重叠，如下，_b 与 _a 地址重叠，需将 _b 定义在 0x0142

  ```
  DEFINE_SFR(unsigned int _a, 0x0140);
  DEFINE_SFR(unsigned char _b, 0x0141); //error
  ```

8.5 变量重定义

如果一个变量 (非绝对地址变量) 定义在头文件，而这个头文件被多个 .c 文件引用，则会出现变量重定义，如下：

```
Linking...
Error(L1031) : Public symbols are duplicated
               Public symbol '_a' in the file C:\Documents and Settings\ydwang\My Documents\HTK_Project\t\T.OBJ
               Public symbol '_a' in the file C:\Documents and Settings\ydwang\My Documents\HTK_Project\t\T1.OBJ
```

解决方式：

不要在头文件中定义变量，若 t.c 与 t1.c 都需要用到 a，则在其中一个文件定义 a，在头文件中声明 extern int a; 即可，如下：

```
//t.C
#include "t.h"
int a;
void main()
{
   a=2;
}
```

```
//t.h
extern int a;
```

```
//t1.c
#include "t.h"
void fun()
{
   a=3;
}
```

第九章 程序范例

本章将使用 C compiler V3 编写常见的 MCU 程序，以便快速使用 C compiler V3 进行工程开发。

涵盖如下主要内容：

- 中断函数的使用
- 混合编程的用法

9.1 使用中断让 LED 灯闪烁

使用定时器让 LED 灯闪烁，时间间隔为 1s，即亮 1s 钟后熄灭 1s 钟。

代码清单 7.1：

```
#include "HT66F50.h"
void main()
{
   _acerl=0x00;
   _cp0c=0x08;
   _pac = 0x00;                  //set PAC as output
   _pa = 0xff;                   //All SEG off
   _mf0e = 0x01;                 //enable Multi-function 0 interrupt
   _t2ae = 0x01;                 //enable T2A interrupt
   _tm2c0 = 0x30;                //set clk = f(sys)/64
   _tm2c1 = 0xc1;                //set Compared with CCRA
   _t2af = 0x00;                 //clear T2A interrupt flag
   _mf0f = 0x00;                 //clear Multi-function 0 flag
   _emi = 1;                     //enable interrupt
   _tm2al = 0x03;                //Matching value
   _tm2ah = 0x00;
   _t2on = 1;                    //start counting
   while(1);
}

DEFINE_ISR(ISR_ADC, 0x14)        //definition ISR
{
   _t2af = 0x00;                 //clear T2A interrupt flag
   _pa = ~_pa;
   _tm2al = 0x24;                //Matching value
   _tm2ah = 0xf4;
}
```

9.2 使用表格让 7 节 LED 管显示数字

这里是用混合编程的形式从汇编表格中读出数据显示在 7 节 LED 管上。

代码清单 7.2：

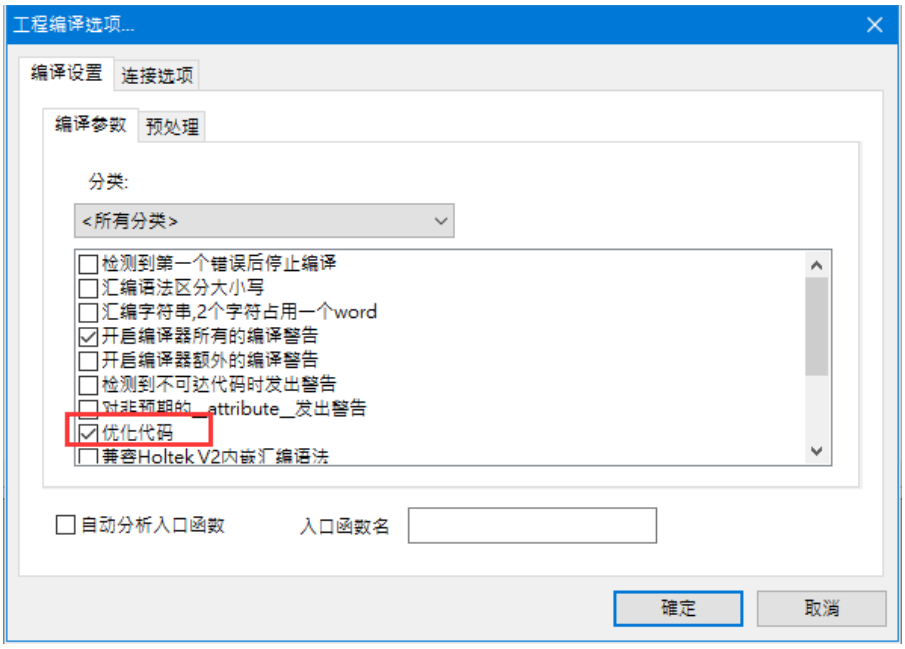
```
Table.ASM
#INCLUDE HT66F50.INC

public _code

_code .SECTION 'CODE'
_code proc
```

```
TAB_7_SEG:
   DC 0F9C0H
   DC 0B0A4H
   DC 09299H
   DC 0F882H
   DC 09580H
   DC 08388H
   DC 0A1A7H
   DC 08E86H
_code endp
   END
Main.c
#include “HT66F50.h”
void _delay(unsigned char times)
{
   volatile unsigned char t1, t2, t3;
   t1 = times;
   while(t1--)
   {
      t2 = 3;
      while(t2--)
      {
         t3 = 110;
         while(t3--);
      }
   }
}
asm(“extern _CODE:near”);
void main()
{
   unsigned char k;
   _acerl=0x00;
   _cp0c=0x08;
   _pac = 0x00;
   _pa = 0xff;
   asm(“mov a, low _CODE”);
   asm(“mov %0, a” : “=m”(_tblp));
   asm(“mov a, high _CODE”);
   asm(“mov %0, a” : “=m”(_tbhp));
   while(1)
   {
      for(k = 0; k < 8; k++)
      {
         asm(“tabrdc %0” : “=m”(_pa));
         asm(“inc %0” : “=m”(_tblp));
         _delay(50);
         asm(“mov a, %0” : “=m”(_tblh));
         asm(“mov %0, a” : “=m”(_pa));
         _delay(50);
      }
      asm(“mov a, 0ffh”);
      asm(“mov %0, a” : “=m”(_pa));//all SEG off
      _delay(50);
   }
}
```

第十章 程序优化写法

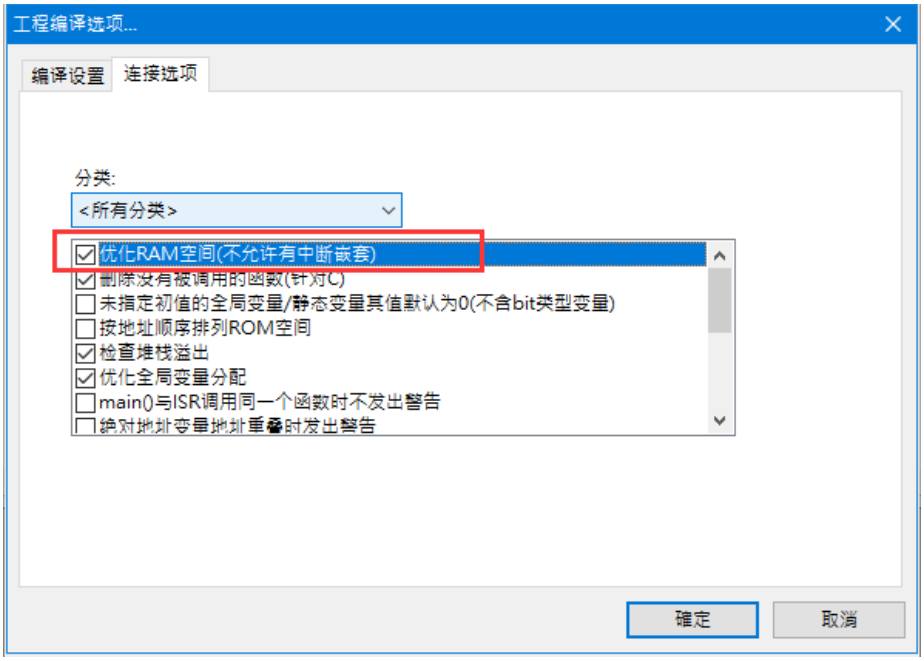
10.1 优化选项

选择“优化代码”，即 -Os，如下图，compiler 会生成较佳的优化代码，具体的优化内容参考第三章内容。

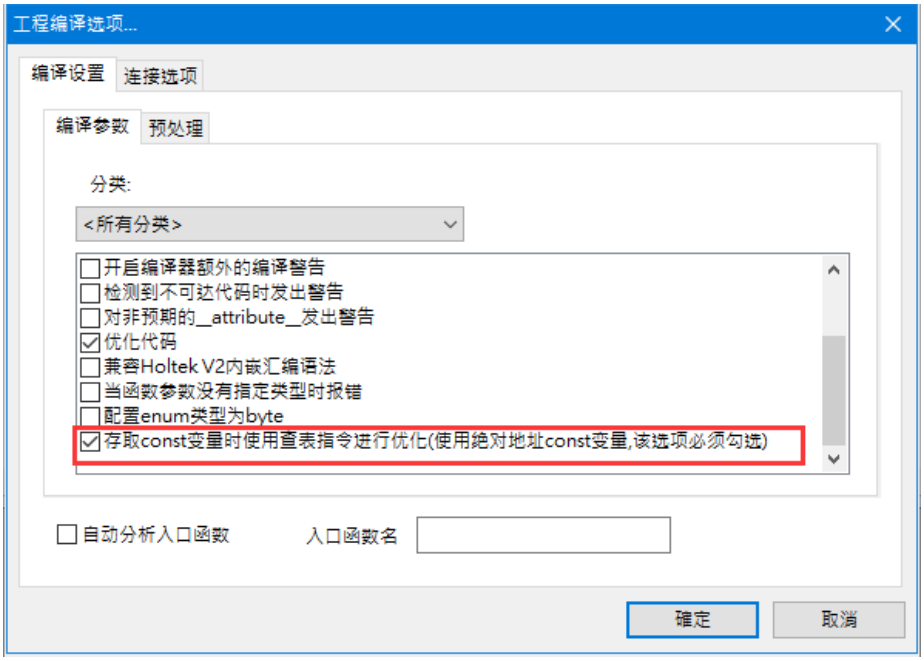


如果整个程序中的中断服务程序没有发生嵌套，即执行一个中断时不会进入另一个中断，则可以选择以下 “优化 RAM 空间” 参数，节省 RAM 空间。

中断的程序要尽量简短，否则会占用太多的 RAM。

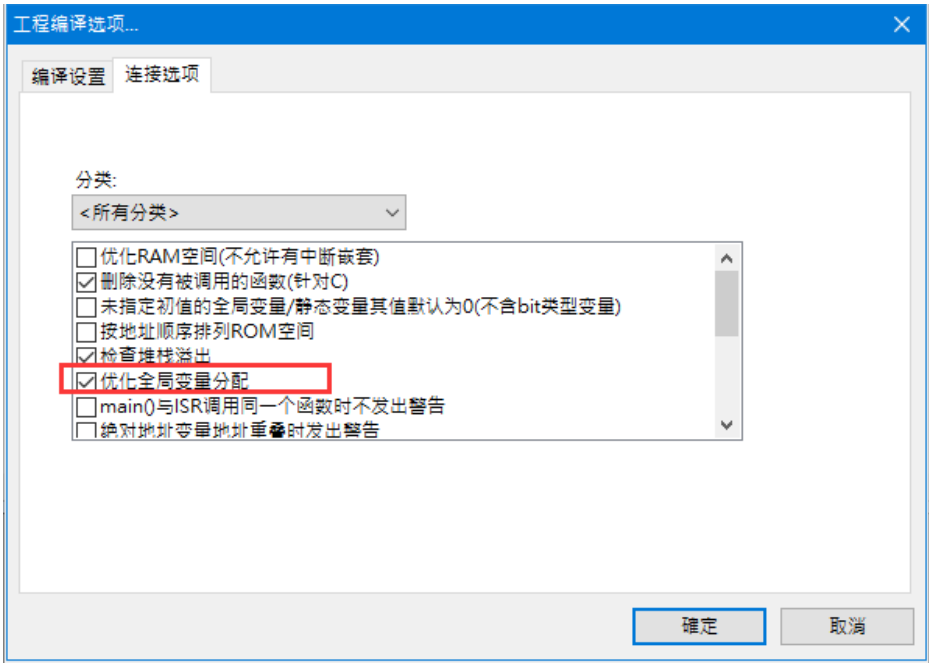


对具有 TBHP 且 ROM 宽度为 16bit 的无扩展指令 MCU，编译程序支持用查表指令的方式存取 const 变量，如下参数 (此参数默认勾选)，如果 const 变量个数很多，勾选此参数会节省 code，反之，则会浪费 code，用户可以根据自己的实际程序选择是否勾选。

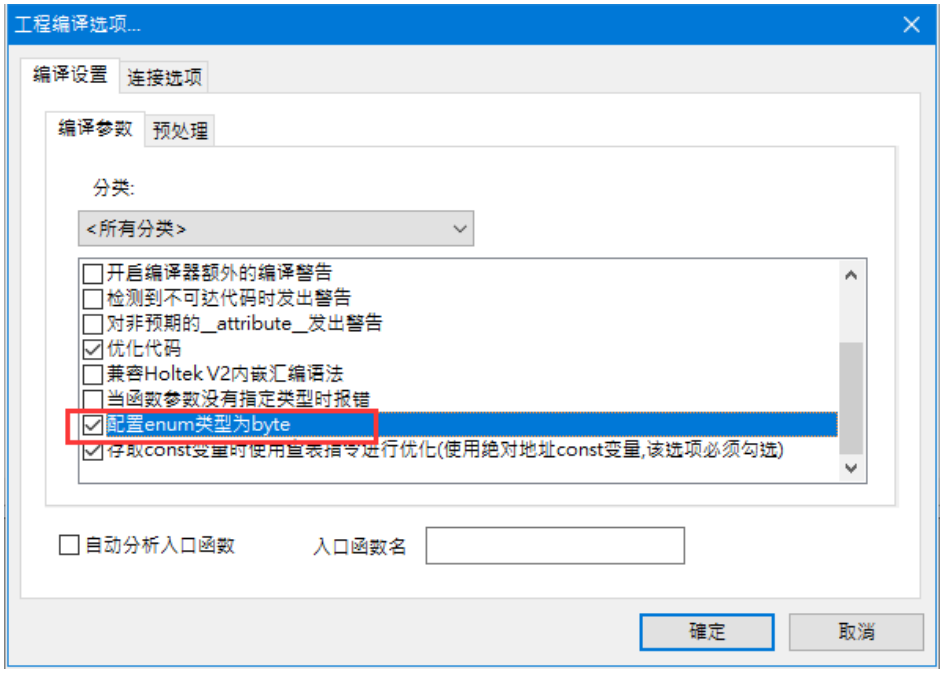


可以选择以下参数“优化全局变量分配”：

具扩展指令架构的 MCU，不需要指定变量地址，连接器会自动分配所有的变量地址 (包括 RAM bank0 以外)，按变量的使用频率分配，优先分配次数多的变量到 bank0。



若定义的 enum 数据不超过 127，或勾选以下选项：



10.2 变量声明

10.2.1

unsigned/signed

在不会出现负数的环境中，使用 unsigned 会更节省 code size。

char array[10];
void main(void)
{
  char i;
  for(i = 0;i<=9;i++)
  array[i] = 0;
}
	char array[10];
	void main(void)
	{
	  unsigned char i;
	  for(i = 0;i<=9;i++)
	  array[i] = 0;
	}

Code size: 34	Code size: 31

10.2.2 数据形态

选择适当范围的数据类型，助于产生更精简的指令

long i;
void main(void)
{
  if(i>=456)
  i = 2;
}
	unsigned int i;
	void main(void)
	{
	  if(i>=456)
	  i = 2;
	}

Code size: 22	Code size: 14

10.2.3 浮点常量

浮点常量默认为 double 类型，如果计算所需要的精度不需太高，可以将它强制转为 float，如：(float)3.14。

```
float s,r;
void main()
{
  s = r * r * 3.14;
}
```
Code size:343

```
float s,r;
void main()
{
  s = r * r * (float)3.14;
}
```
Code size:177

编译程序不会给浮点运算常量折叠，所以，如果有两个浮点常数需要计算，可以把结果先算出来：

```
#define HALF (float)0.5
#define QUARTER (float)0.25
float l,r;
void main()
{
  r = l * HALF * HALF;
}
```
Code size:171

```
#define HALF (float)0.5
#define QUARTER (float)0.25
float l,r;
void main()
{
  r = l * QUARTER;
}
```
Code size:152

10.2.4 const 数组

把 const 数组定义成全局会比局部的节省 RAM：

```
unsigned char sum;
unsigned char dx[7];
void main()
{
  const unsigned char tx[7] =
  {1,3,5,15,5,3,1};
  unsigned char i;
  for(i=0;i<7;i++)
    sum += dx[i]*tx[i];
}
```
RAM size:23

```
const unsigned char tx[7] =
{1,3,5,15,5,3,1};
unsigned char sum;
unsigned char dx[7];
void main()
{
  unsigned char i;
  for(i=0;i<7;i++)
    sum += dx[i]*tx[i];
}
```
RAM size:16

10.2.5 将功能相似的变量定义成数组以便使用循环语句

```
unsigned int n1,n2,n3,n4;
void func()
{
  unsigned char i;
  for(i=0; i<10; i++)
  {
    n1 += (i * 201);
    n2 += (i * 202);
    n3 += (i * 203);
    n4 += (i * 204);
  }
}
```
Code size:140

```
unsigned int n[4];
void func()
{
  unsigned char i,j;
  for(i=0; i<10; i++)
  {
    for(j=0; j<4; j++)
    n[j] += (i * (j + 200));
  }
}
```
Code size:75

10.2.6 除 delay 函数外，局部变量不使用 volatile 修饰

10.3 程序结构

10.3.1 调整语句顺序以适用尾部合并优化

编译器会做尾部合并优化，参考 3.12 节，对于 switch 或 if/else 语句，可以将相同的语句写在分支的后面，以适用尾部合并。

```
unsigned char n;
unsigned char array[4];
void func()
{
  switch(n)
  {
    case 1:
      array[1] = 0xff;
      array[0] = 4;
      array[2] = 2;
      array[3] = 1;
      break;
    case 2:
      array[2] = 0xff;
      array[0] = 4;
      array[1] = 3;
      array[3] = 1;
      break;
    case 3:
      array[3] = 0xff;
      array[0] = 4;
      array[1] = 3;
      array[2] = 2;
      break;
    default : break;
  }
}
```

Code size:33

```
unsigned char n;
unsigned char array[4];
void func()
{
  switch(n)
  {
    case 1:
      array[1] = 0xff;
      array[2] = 2;
      array[3] = 1;
      array[0] = 4;
      break;
    case 2:
      array[2] = 0xff;
      array[3] = 1;
      array[1] = 3;
      array[0] = 4;
      break;
    case 3:
      array[3] = 0xff;
      array[2] = 2;
      array[1] = 3;
      array[0] = 4;
      break;
    default : break;
  }
}
```

Code size:29

10.3.2 重复多次的运算可以用循环代替

当程序中存在多次重复的运算，而且都有规律性，可以用循环代替。

```
unsigned char show_data[6];
unsigned long hex;
void main()
{
  show_data[5]=hex%10;
  show_data[4]=hex/10%10;
  show_data[3]=hex/100%10;
  show_data[2]=hex/1000%10;
  show_data[1]=hex/10000%10;
  show_data[0]=hex/100000%10;
}
```

Code size: 378

```
unsigned char show_data[6];
unsigned long hex;
void main()
{
  unsigned long temp=hex;
  unsigned char i;
  for(i=6;i>0;)
  {
       i--;
       show_data[i]=temp%10;
       temp/=10;
  }
}
```

Code size: 156

10.4 函数调用

10.4.1 避免不必要的函数调用

若某函数被多次调用，但其返回值并无差异，应考虑用变量接收函数返回值，使用时做为替代。

int fun(int i)
{
  return i;
}
int i;
void main(void)
{
  if(fun(i)== 0)
  i = 0;
  else if (fun(i) == 1)
  i = 6;
  else if (fun(i) == 2)
  i = 4;
}
	int fun(int i)
	{
	  return i;
	}
	int i;
	void main(void)
	{
	  int temp = fun(i);
	  if(temp== 0)
	  i = 0;
	  else if (temp == 1)
	  i = 6;
	  else if (temp == 2)
	  i = 4;
	}

	减少运行时间

10.4.2 封装频繁使用的代码为函数

如果在程序中存在某段代码被多次使用，可将这段代码封装成为函数，以便节省指令。

char array[10][10];
void func1()
{
  unsigned char i,j;
  for(i = 0;i <= 9; i++)
  for(j = 0;j<= 9; j++)
  array[i][j] = 0;
}
void func2()
{
  unsigned char i,j;
  for(i = 0;i <= 9; i++)
  for(j = 0;j<= 9; j++)
  array[i][j] = 0xff;
}
	char array[10][10];
	void init_array(char n)
	{
	  unsigned char i,j;
	  for(i = 0;i <= 9; i++)
	  for(j = 0;j<= 9; j++)
	  array[i][j] = n;
	}
	void func1()
	{
	  init_array(0);
	}
	void func2()
	{
	  init_array(0xff);
	}

Code size:108	Code size:52

10.4.3 如果函数只在本文件调用，可以定义成 static

float s;
unsigned char func(unsigned char *dx)
{
    unsigned char sum;
    sum = dx[0]+ dx[1]+ dx[2] ;
    return sum;
}
unsigned char sum;
unsigned char array[4];
void main()
{
    sum = func(array);
    s = sum * (float)3.14;
}
	float s;
	static unsigned char func(unsigned char *dx)
	{
	    unsigned char sum;
	    sum = dx[0]+ dx[1]+ dx[2] ;
	    return sum;
	}
	unsigned char sum;
	unsigned char array[4];
	void main()
	{
	    sum = func(array);
	    s = sum * (float)3.14;
	}

Code size:222	Code size:184

10.5 全局变量的分配

对无扩展指令架构的 MCU，RAM BANK0 以外的地址只能用间接寻址的方式访问，下例可以看出，间接寻址的指令比直接寻址多出 5 条，所以，当 RAM bank0 溢出时，用户可以选择把较少用到的变量定义到其它 bank，比较常用的变量定义在 bank0。

直接寻址 (Bank 0)	间接寻址 (非 Bank 0)
Rambank 0 ds ds .section 'data' _var0 db ?	Rambank 1 ds ds .section 'data' _var1 db ?

MOV A,40H
MOV _var0, A
	MOV A,BANK _var1
	OR A,ROM_BANK FUNC
	MOV BP,A
	MOV A,OFFSET _var1
	MOV MP1,A
	MOV A,40H
	MOV IAR1,A

Code size: 2 words	Code size: 7 words

10.6 中断服务程序

一般，如果两个函数没有调用关系，那它们的局部变量是可以分配到同样的地址，但中断服务程序不会与主函数共享局部变量的地址，所以，为了减少 RAM 的使用，中断程序可以尽量简单，不宜写得太过复杂。

10.7 变量初始化

若程序中已经有写 CLR RAM 的函数，可将全局 / static 变量定义的初始值去掉，因为此时初始值无效。

附录 A：ASCII 码表

DEC	HEX	Symbol	DEC	HEX	Symbol	DEC	HEX	Symbol	DEC	HEX	Symbol
0	0	NUL	32	20	space	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	

附录 B：运算优先级

优先级	运算符	含义	运算类型	结合性
1	( )	圆括号	单目	自左向右
	[ ]	下标运算符		
	->	指向结构体成员运算符		
	.	结构体成员运算符		
2	!	逻辑非运算符	单目	自右向左
	~	按位取反运算符		
	++ --	自增、自减运算符		
	(类型关键词)	强制类型转换		
	+ -	正、负号运算符		
	*	指针运算符		
	&	地址运算符		
	sizeof	长度运算符		
3	* / %	乘、除、求余运算符	双目	自左向右
4	+ -	加、减运算符	双目	自左向右
5	<<	左移运算符	双目	自左向右
	>>	右移运算符		
6	< <= > >=	小于、小于等于、大于、大于等于	关系	自左向右
7	== ! =	等于、不等于	关系	自左向右
8	&	按位与运算符	位运算	自左向右
9	^	按位异或运算符	位运算	自左向右
10	|	按位或运算符	位运算	自左向右
11	&&	逻辑与运算符	位运算	自左向右
12	||	逻辑或运算符	位运算	自左向右
13	? :	条件运算符	三目	自右向左
14	= += -= *= /= %= << = >>= &= ^= |=	赋值运算符	双目	自右向左
15	,	逗号运算符	顺序	自左向右

附录 C：命令行模式命令参数及功能

1、compiler 参数

命令：hgcc32 [options] cfile –o asmfile

参数	含义
-g	产生 debug 信息
-O0/-O1/-O2/-O3/-Os	优化参数
-D<macro>[=<val>]	宏定义
-I<path>	设定搜寻头文文件的目录 path
-msingle-ram-bank	单个 RAM
-mmulti-ram-bank	多个 RAM(default)
-msingle-rom-bank	单个 ROM
-mmulti-rom-bank	多个 ROM(default)
-fno-builtin	不使用 gcc 内建函数
-mno-tbhp	没有 TBHP
-mtbhp=addr	指定 TBHP 地址 addr(default 为 1fH)
-mlong-instruction	扩展指令 MCU

2、assembler 参数

命令：hasmgcc32 [options] source, object, listing

参数	含义
/chip=chip-name	指定 MCU 型号
/case	区分大小写
/d<macro>	宏定义
/i<include-path>	设定搜寻头文文件的目录 path
/z	产生 debug 信息
/h (/?)	显示帮助
source	要编译的 asm 文件
object	指定生成的 OBJ 文件的名字
listing	指定生成 lst 文件的名字

3、linker 参数

命令： /HIDE=xxx /MCU=xxx [/NOLOGO] [/novectornest] [/OptimizeParam=x] [/OptimizeLInst=x] [/Startup0] [/EEPROM=xx] [/TBHP=x] [/ERRORLOG="xxx"] [/option] objectfile [,taskfile [,mapfile [,dbgfile [,libraryfiles]]]] [;]

注意：以上的参数一定要按照上面的顺序设置，否则 linker 会出错或不能正确执行，而且参数是区分大小写的。

环境变量 LIB 为搜寻 lib 文件的目录

参数	含义
/HIDE	IDE 的句柄。8 个 16 进制数
/MCU	工程使用的 MCU 名称
/NOLOGO	隐藏 LOGO
/novectornest	有该参数表示 ISR 不会嵌套， 可以节省 RAM space (for C Compiler V3 only)
/OptimizeParam=x	默认为 0， x 的低半字节表示 BP 优化参数，0 表示关闭， 2 表示开启 x 的高半字节表示 Dead Section 优化，0 表示关闭， 1 表示开启 (for C Compiler V3 only)
/OptimizeLInst=x	表示扩展指令优化，0 表示不优化，1 表示优化。 默认不优化 (for C Compiler V3 only)
/Startup0	将没有初始值的全局变量初始化为 0 (for C Compiler V3 only)
/TBHP=x	x 表示 TBHP 寄存器的地址，默认地址为 9
/EEPROM=xx	指定 EEPROM_DATA_SIZE，默认为 0 (for C Compiler V3 only)
/ERRORLOG	保存错误日志的路径
/MAP	用户指定必须生成 MAP 文件
/ADDR:section_name=addr [,section_name=addr]	指定某些 section 的分配地址由指定地址 address 开始。 注：addr 为 16 进制
/HELP (/?)	显示命令行格式帮助信息

注意：调用 assembler 和 linker 前先设置环境变量 CFG。

set HTCFG=IDE 目录 \ MCU。

参考读物

《HT-IDE3000 使用手册》

介绍 HT-IDE、assembler、linker 等 tool 的使用，可于 HT-IDE3000 安装文件 DOC 目录下取得。

《Holtek 标准函数库使用手册》

介绍 Holtek C 支持的标准函数库及使用方式，可于 HT-IDE3000 安装文件 DOC 目录下取得。

《Holtek C Compiler V3 FAQ》

C Compiler V3 常见的 FAQ 问题，持续更新中，可于 HT-IDE3000 安装文件 DOC 目录下取得。

《gcc manual》

GCC 使用手册，可至 http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc.pdf 下载。