C Compiler V3 使用手冊

版本：V2.00　日期：2024-03-22

www.holtek.com

用戶須知

所有文檔均會過時，本文檔也不例外。Holtek 的工具和文檔將不斷演變以滿足客戶的需求，因此實際使用中有些對話方塊和工具說明可能與本文檔所述之內容有所不同。請訪問我們的網站：

(http://www.holtek.com.tw/mcu_tools_users_guide) 獲取最新文檔。

目錄

前言

本手冊主要講述了 C 語言的基礎語言，再以此為基礎，進而講述 C compiler V3 的語法結構和其優化功能，幫助程式員快速使用 C compiler V3 開發應用程式。

C compiler V3 是由 GCC 4.6.2 以上版本移植過來的，除後端輸出，其餘部份都可參考 GCC 與機器無關的相關使用手冊。

這裏假定讀者已具備如下基本素質：

- 知道如何編寫 C 程序
- 已經閱讀并理解所使用單片機的數據手冊

第一章 C 語言基礎知識

本章將由淺到深的概括 C 語言的基礎語法及結構特點，方便後面學習 C compiler V3，由於受限於單片機的硬體結構，因此本章的描述基於標準 C 語言，相容 C compiler V3 之語法。

主要包含如下內容：

- 數據類型、運算符與表達式
- 函式
- 陣列與指標
- 結構體、聯合體與枚舉
- 預處理
- 流程控制
- 作用域

1.1 數據類型、運算符與表達式

1.1.1 C 的數據類型

數據類型確定了變數在內存中佔用的存儲單元，所以在宣告變數時首先必須要確定變數的類型，數據類型可以分為基本數據類型、構造數據類型、指標類型 (Pointer) 和空類型 (void)，基本數據類型有整型、字元型、浮點型，構造類型則有陣列、結構體、共用體和枚舉，利用這些構造類型可以構造出所需要的數據結構。

列舉基本數據類型 (C compiler V3) 如表 1-1-1：

數據類型	佔用空間 (bit)	範圍
bit[1]	1	0, 1
_Bool[2]	8	0, 1
char [3]	8	-128 ~ 127
unsigned char	8	0 ~ 255
short	16	-32768 ~ 32767
unsigned short	16	0 ~ 65535
int	16	-32768 ~ 32767
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 基本數據類型 (C compiler V3)

註：[1]、bit 类型，最低位有效。比如 bit a = 4；則 a = 0。
[2]、_Bool 類型，當值非 0 時為 1，當值為 0 時則為 0。比如 _Bool a = 4；則 a = 1。
[3]、基本數據如果沒有 unsigned 修飾，則默認為 signed，下同

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

符號 sign	指數 e	尾數 m
23	22~15	14~8 7~0

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

符號 sign	指數 e	尾數 m
31	30~23	22~16 15~8 7~0

1.1.2 常數與變數

在程式運行過程中，常數是其值不能被改變的量，與之對應的是變數，變數是在內存中具有特定屬性的一個存儲單元，它用來存放數據，根據所需要存放數據的數據類型定義相應的變數。

1.1.2.1 數字常數、字串常數、const 常數

- 數字常數：C Compiler V3 支援指定十六進位 (0x) 和八進制 (0) 值的標準前綴，另外還支援用前綴 0b 來指定二進制值。例如，數值 237 可以表示為二進制常數 0b11101101。
- 字串常數：一般用“”引起來的符號串，如“abc”。
- const 常數：C Compiler V3 支援最大 64Page 的 const 常數，且支援 const 陣列，指向 const 的指標等。

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);
}
```

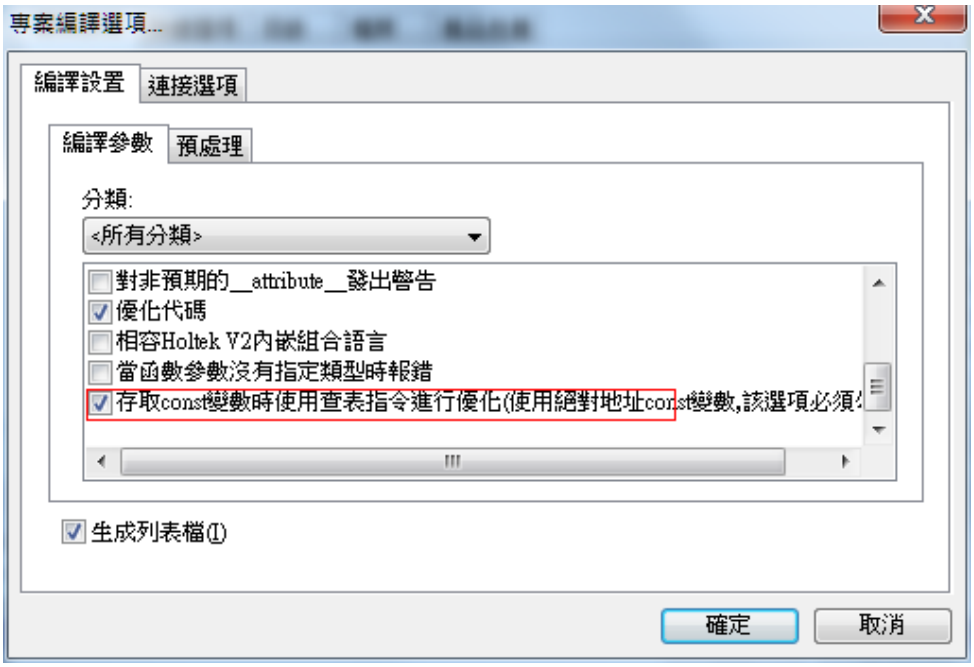
對有擴展指令的 MCU (如 HT66F70A) 或具 TBHP 寄存器且 PROM 寬度為 16bit，V3 Compiler (v3.20 以上版本) 支援指定位址的 const 變數，其語法如下：

```
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 無擴展指令時 , 需勾選以下參數：



如此，ci 在 ROM 中的存放結果如下：

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

1.1.2.2 變數及其定義

變數是在程式運行時，其值可以變化的，每個變數都應該有一個名字，以便被引用，並區分大小寫，C 語言規定，所有的變數都必須先宣告，後使用，變數在定義時必指定數據類型，這樣編譯時才可為其分配對應的存儲單元。如 int a;。

標識符是用來標識變數、常數、函式、類型等的字元系列，C 語言規定標識符只能由字母、數字、下劃線構成，且必須以字母和下劃線作為起始符。

1.1.2.3 變數的存儲方式

在 C 語言中每個變數和函式都有兩個屬性：數據類型和數據的存儲方式，存儲方式可分為 2 種，靜態存儲和動態存儲，具體又包含 4 種，自動的 (auto)、靜態的 (static)、寄存器的 (register)、外部的 (extern)。

1、auto：函式中的局部變數，如果不專門聲明為 static 的存儲方式，則默認為 auto，所以在函式內 auto char a 與 char a 是等價的。

2、static：可分為全域靜態存儲和局部靜態存儲，全域變數加了 static 後，變數只能在本檔中引用，局部靜態存儲則是局部變數的值在函式調用結束後不消失而保留原值，在下一次調用該函式時，該變數已經有值了。

3、register：上述兩種變數是存放在內存中的，而 register 則是把變數存放在寄存器中，基於單片機的特殊情況，這裏不展開敘述。

4、extern：使用另一個檔中定義的變數，表示該變數是一個已經在外部定義過的變數，只要加上 extern 就可以使用該變數了，後面有專門的講述。

5、volatile：一個類型修飾符 (type specifier)。它是被設計用來修飾被不同程式訪問和修改的變數，使用 volatile 修飾的變數，不會因編譯器的優化而被省去。

建議定義成 volatile 的變數：特殊寄存器，中斷函式使用到的變數，為某些特殊用途的代碼定義的變數 (比如 delay 功能)。

其它一般變數不建議定義成 volatile，這樣會大大降低編譯器的優化功能。

代碼清單 1.1：

```
File1.c
int cpv;

File2.c
extern cpv;              // 引用 File1.c 中的 cpv 變數
int c;
int statictest()
{
   cpv = 5;
   static int k = 26;    // 靜態變數
   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
}
```

兩次運行的結果是 34，35

1.1.3 C 語言運算簡介

C 語言的運算十分豐富，主要包括算術運算，邏輯運算、位運算、賦值運算、條件運算、逗號運算等，各種運算及其之間的優先級見附錄 B。

1.1.3.1 類型轉換

類型轉換規則：

- 混合類型的算數運算
  - ♦ 小于 int 類型的轉換為 int 類型
  - ♦ 小類型向大類型轉換 (轉換過程見圖 2_1_1)
- 不同類型之間的賦值
  - ♦ 以賦值語句左邊的類型為轉換後類型
- 函式參數 / 返回值的傳遞
  - ♦ 以參數 / 返回值的類型為轉換後類型

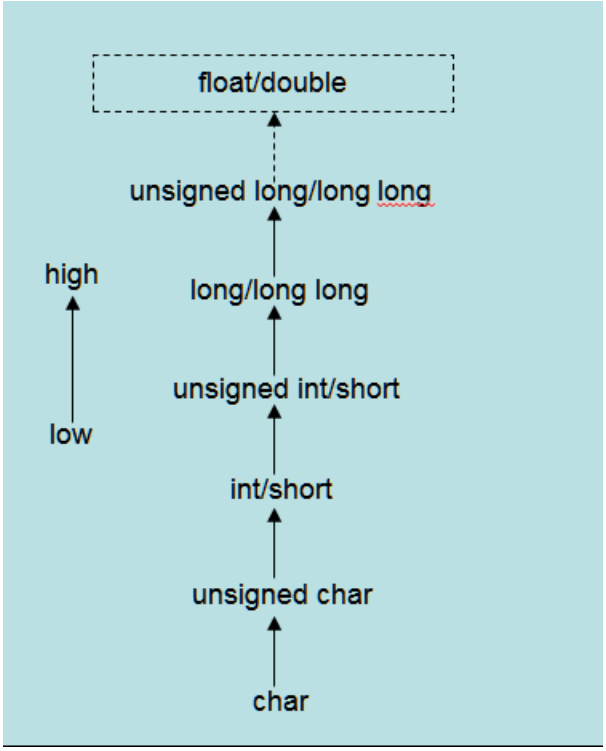


圖 2_1_1

類型轉換示例：

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;
	}

結果：C = -25536; 說明：a 與 b 都是 int 類型，使用 int 計算，結果為 40000，超出 int 的範圍，所以結果為 -25536。 解決方式： long c = (long)a+b; 結果：c =40000;	結果：c = 200; 說明：a 與 b 都 <int，提升為 int 計算，結果為 200

1.1.3.2 邏輯運算

邏輯運算和關係運算的結果只有兩個：真、假，如果運算出來的結果為真，則用邏輯 1 來表示，反之則為邏輯 0，邏輯運算就是把各關係運算或其它邏輯量進行與 (&&)、或 (||)、非 (!) 的運算。

邏輯運算具有短路性。

代碼清單 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)){           // 邏輯或的短路功能
           e = 0x18;
    }
    else{
           e = 0x81;
    }

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

運算結果：a = 0x41，b = 0x31，c = 0x01，d = 0x55，e = 0x18，f = 0x81，
g = 0x00，h = 0xa5，d 和 h 沒有被賦值。

非運算 (!) 的特殊應用：!!(0xXX)，當 0xXX 不為 0 時，兩次取非運算的結果則為 1。

1.1.3.3 位運算

C 語言中兩個很具魅力的地方，一個是指標，另一個是位運算，本小節講述的便是 C 語言中的位運算，C 語言中位運算共有 6 種，按位與 (&)、按位或 (|)、按位異或 (^)、取反 (~)、左移 (<<) 和右移 (>>)，巧妙利用位運算可以簡化很多運算時間。常見應用及操作：

1、把小寫字母變為大寫字母，清位元：‘a’ & 0xDF，結果為‘A’

2、把大寫字母變為小寫字母，置位元：‘A’ | 0x20，結果為‘a’

3、對某位取反，某個位與 1 異或即為取反 (第 1 位取反)：0xFF ^ 0x01，運算的結果為 0xFE

4、部分乘法的化簡，與 2 的 n 次方相乘，相當於左移 n 位，例如 0x02 乘以 4，0x02 << 2，這裏的 2，表示 4 = 2 的‘2’次方，結果為 8

5、部分除法的化簡，與 2 的 n 次方相除，相當於右移 n 位，例如 0x08 除以 4，0x08 >> 2，這裏的 2，表示 4 = 2 的‘2’次方，結果為 2

6、部分求餘的化簡，與 2 的 n 次方求餘，跟 2 的 n 次方 -1 與，如 15 跟 8 求餘，相當於 15 & 7，這裏的 7 是 8-1 = 7，結果為 8

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

8、循環移位，對一個 16 位的數循環左移 n 位，0xXX >> (16 - n) | 0xXX << n

9、循環移位，對一個 16 位的數循環右移 n 位，0xXX << (16 - n) | 0xXX >> n

1.2 函式

函式是相當於副程式，每一個函式都可用來實現一個特定的功能。

在程式開發的過程中，常使用的功能模塊編寫成函式，以減少重覆編寫程式段的工作量，函式的本質是一個標號，即一個地址，調用該函式，相當於跳轉到這個位址去執行指令。

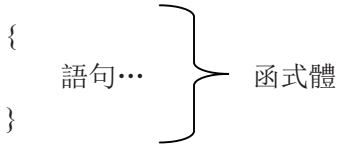
1.2.1 函式的聲明與定義

函式聲明時，不必先定義函式，只是做一個聲明。聲明的函式必須是存在的函

式，如果使用庫函式，則直接把頭檔包含進來。如果是用戶自定義的函式，則是聲明函式將在該檔後面定義，在函式內不能再定義函式，即不能嵌套定義。

函式的定義包括返回值數據類型，函式名，參數列表和函式體，形式：

返回值數據類型 函式名 (參數列表)



例子 2：求兩個數的較大者。

代碼清單 1.3：

```
int max(int, int);        // 函式聲明
int a;
void main()
{
   a = max(10, 20);       // 函式調用
   a++;
}
int max(int a, int b)     // 返回值類型 函式名 ( 參數列表 )
{
   return a > b ? a : b;
}
```

運算結果 a = 21

1.2.2 參數列表與返回值

參數列表，函式名之後，用括號括起來，在調用函式時要傳遞給函式的變數列表稱為參數列表。

關於形參與實參的說明：

1、實參可以是常數、變數或表達式，但要求具有確定的值，在函式調用時，把值賦給形參。

2、函式的定義時，必須指定形參的數據類型。

3、實參與形參必須要相容。

4、形參的最大特點是單向值傳遞，即只是把值傳過來，而並沒有改變實際參數的值。

5、在 C compiler V3 中，參數及函式局部變數的命名方式是 _funname_2[n]，比如：函式 fun 的變數命名為 _fun_2，n 則表示它共有多少個變數。

6、若返回值為 1byte，則存於 ACC，若為 2byte，則低字節存於 ra，高字節存於 rb，若為四 byte，由低字節到高字節分別存於 ra、rb、rc、rd。

7、由於 MCU 的限制 (沒有堆棧)，參數和內部變數會佔用 RAM 空間，而互不影響 (沒有調用關係) 的函式變數可以共用同樣的 RAM 空間。

例子 3：可以改變 a 的值嗎？

代碼清單 1.4：

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

```
{
   a = 15;
   change(a);
}
```

運算結果 a = 15

返回值是把運算的結果返回給該函式的調用者使用，void 類型的函式是否可以使用 return 呢？可以的，只是 return 後面不要加任何表達式。

1.2.3 函式的調用

1.2.3.1 函式的調用方式及過程

函式的調用方式有 3 種：

1、函式語句：只把函式調用作為一個語句，完成一定的功能，如 change(a);

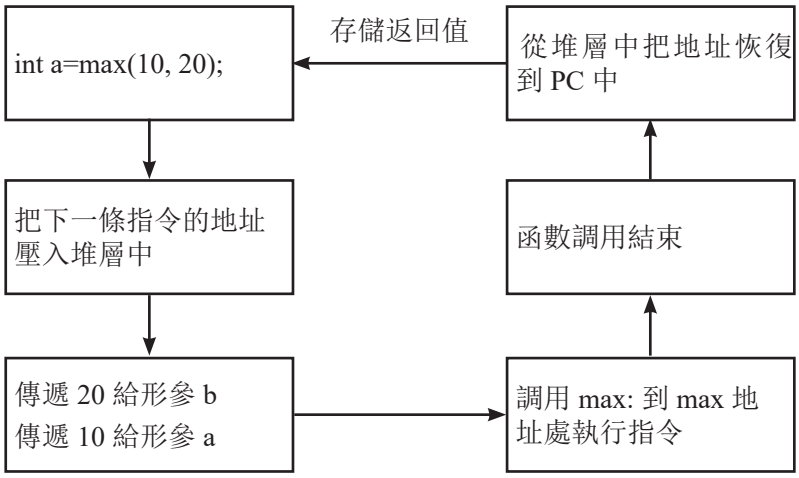
2、函式表達式：如 int a = 3*max(10, 20);

3、函式參數：如 int a = max(max(20, 30), 20);

函式的調用過程：

不同的編譯器其函式調用過程不盡相同，函式參數的傳遞的方向也不相同，在 C 語言中，形參是從右到左的傳遞，下面講述 C compiler V3 下函式調用過程 (以例子 2 為例子)：

流程 1：



函式區域變數的位址分配：

1、HT8 MCU 沒有 RAM stack, 所有函式的區域變數都是編譯時 (link 階段) 分配好的，佔用 RAM 空間

2、Linker 根據函式呼叫關係分配區域變數，若 function1 與 function2 無調用關係，則 function1, function2 可以共用區域變數空間 , 有調用關係則無法共用。

3、linker 為主程式跟中斷各分配獨立的區域變數空間，不會共用 (也不允許中斷與主程式共同調用子函數)。

4、除了區域變數，編譯器還會用到 8 個中間變數 ra~rh，因為沒用通用寄存器，所以 ra~rh 也是分配在普通 RAM 裡面，同樣主程式跟中斷的 ra~rh 也是獨立分開分配。

1.2.3.2 嵌套調用

函式的嵌套調用是指在函式內調用別的函式，函式是可以嵌套調用的，但是不可以嵌套定義，而遞歸調用則是特殊的嵌套調用，是指一個函式內又直接或間接調用該函式本身。

1.2.3.3 外部函式的使用

使用外部函式有兩種方式，一種是通過頭檔包含，一種則是使用 extern 關鍵字。

1、頭檔包含的方式：

例子 4：調用外部加，減函式。

代碼清單 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);
}
```

執行結果：a = 10, b = 10, c = 20, d = 10

2、extern 的方式：

例子 5：調用外部乘，除函式。

代碼清單 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);
}
```

執行結果：a = 10, b = 10, c = 100, d = 1

1.2.3.4 內聯函式

使用關鍵字 inline 修飾函式，則該函式為內聯函式，內聯函式直接把函式體的語句替換函式的調用。以節省函式調用時因保護現場的時間代價和使用堆層的空間代價。多次調用時會大量增加代碼。

代碼清單 1.7：

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

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

這裏直接把 max = var2 > var1 ? var2 : var1; 這句代碼代換了 getmax(23,32); 函式。

1.2.4 main 函式

main 函式是特殊的函式，程式運行時的入口函式，當初始化完單片機的環境之後，接下來就執行 main 函式中的語句。所以每個工程都應該有一個 main 函式，main 函式不需要參數及返回值。

1.2.5 標準函式庫

C Compiler V3 (v3.20 以上版本) 支援 math.h、string.h 等標準函式庫，其使用說明詳見 IDE/doc--< 標準函式庫使用手冊 >。

1.3 陣列與指標

陣列是有序數據的集合，陣列中的每一個函式都屬於同一種數據類型，使用陣列名和下標可以唯一確定陣列中的一個元素。內存區中每個字節都有一個編號，這個編號稱為地址，指標變數的值就存放這些編號。

1.3.1 陣列的定義、初始化與使用

陣列的定義：數據類型 陣列名 [元素個數]，元素個數只能是直接常數和符號常數。如 unsigned char led_table[5] [1]。

陣列的初始化方式：

1、定義一個陣列，如 unsigned char led_table[5] [2]；然後逐個元素賦值。

2、定義的同時初始化，如 unsigned char led_table[5] = {0, 1, 2, 3, 4};

3、不指定個數的初始化，如 unsigned char led_table[] = {0, 1, 2, 3, 4}; 這時會確

定元素個數為 5 個。

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

註：[1]、陣列下標由 0 開始，表示第一個元素，最後一個元素為個數 -1。

[2]、逐個賦值時，陣列索引不能超過元素個數，如本處元素個數為 10，則下標最大為 9，超過會溢出。

[3]、led_table[0] = 2，led_table[1] = 1 其他沒有被賦值的值為 0。

使用時，直接用陣列名加索引號，如 led_table[2]，由於 C 語言不檢查陣列界限，所以在使用時要註意陣列下標溢出的問題。

1.3.2 多維陣列

有多個下標索引，定義並初始化時，最高維不用指定，例如：led_table[][4] = {{1, 2, 3},{},{1}}；本例中第二維已經確定為 3，使用時，每個維數的下標都要指定。

1.3.3 字串及其結束標誌

字串常數存放在 ROM 中，以‘\0’結束，例如，字串 char c[] =“chip”，所以“chip”實際佔用了 5 個字符空間。

如果一個字元陣列，最後一個元素為‘\0’也可當字串使用。

陣列名的本質是陣列在內存空間中的首地址，例如 led_table[5]，則 led_table 為 led_table[0] 的地址。

1.3.4 指標的概念

在程式中定義的變數，會在內存中分配相應的內存單元，而該內存單元有個編號，這就是“地址”，指標存放的內容就是該內存單元的編號，也就是地址。

1.3.5 指標的類型

指標可以指向不同類型變數的位址 (RAM 位址)，比如 int、char 等，所以在定義指標時必須指定指標要指向的數據類型，特殊的指標類型如指向函式的指標，指向指標的指標，指向多維陣列的指標。指標的大小與指標的類型無關，不管是何種類型的指標，其值都固定的且只與體系結構有關，如 char *p;long *q 而使用 sizeof(p) 和 sizeof(q) 時，其值都為 2(C compiler V3)。

1.3.6 指標的操作

操作指標分 3 步：定義指標，初始化指標，使用指標。特別要註意的是在使用指標時必須初始化，如果沒有初始化指標就向指標指向的地址寫入數值會引起不可預見的錯誤。指標的操作符有兩個：&(取位址操作) 和 *(取內容操作)。

1、操作指向變數的指標：

例子 6：可以改變 a 的值嗎？

代碼清單 1.8：

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

運行結果：a = ‘b’，由於傳進來的參數是 a 的地址，改變該地址裏面的值就相當於改變實參的值。

2、操作指標的指標：

例子 7：可以改變 a 的值嗎？

代碼清單 1.9：

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

運行結果：a = 0x80

由於參數的傳遞是單向的，所以如果要改變 a 的值，則改為如下代碼：

代碼清單 1.10：

```
char *a;
void change(char **b)
{
   *b = (char *)0x81;  // 修改指標內容
}
void main()
{
   a = (char *)0x80;
   change(&a);
}
```

運行結果：a = 0x81，

3、操作陣列指標和指標陣列：

陣列指標是指指向陣列的指標，定義形式如：char (*b)[5];

指標陣列則是存放數據類型為指標的陣列，定義形式如：char* a[5]。

有什麼區別呢？在這裡 a 是陣列的首位址，在陣列 a 中存放的是 5 個指標值，而 b 是指標，它指向的是長度為 5 個 char 型的陣列的首位址，如果用 sizeof 運算符便知 a 的長度為 10，而 b 的長度為 2。

4、溢出的下標：

代碼清單 1.11：

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

運行結果：如果 a8 在存放在內存為 0x0085 的位置，則 0x008a 地址的值為 8。如果此時 0x008a 存放一個很重要的數據，則影響了整個程式的運行結果。

5、操作常數指標和指標常數：

常數指標表示指標指向的內容是常數，定義方式如：const char *c_p1;

指標常數表示指標本身的值不能變，但指向的內容可以變，定義方式如：char *const p2_c;

代碼清單 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';            // 企圖修改字串常數的內容，報錯。
   p = "Hello! World!";       // 可以修改指向常數的指標。
   p = b;                     // 也可以指向非常數位址。
   *(q + 1) = 2;              // 可以修改指標常數的指向的內容。
   q = b;                     // 企圖修改常數指標的值，報錯。
}
```

6、指標運算：

指標可以進行加、減、自增、自減等操作，

代碼清單 1.13：

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

運算結果：p = 0x88，a[2] = 10

1.3.7 陣列名與指標的區別

陣列名的本質是該陣列在內存中的首地址，與指標的概念神似，但陣列名和指標卻有絕對的不同之處：

1、陣列名是常數，不可修改，相當於指標常數；

2、sizeof 運算結果不同，陣列名：佔用的空間，指標：固定為 2；

3、陣列名不可自增、自減等。

1.4 結構體、聯合體與枚舉

結構體是數據的集合，與陣列不同之處是陣列中的每個元素必須是同種類型，而結構體沒有這個限制；聯合體與結構類似；枚舉則是把變數的值限制在一個指定的範圍內。

1.4.1 結構體、聯合體與枚舉的使用

例子 8：寄存器 PA 和 PA5 的定義 (利用結構體和聯合體)

代碼清單 1.14：

```
#define DEFINE_SFR(sfr_type, sfr, addr) \
static volatile sfr_type sfr __attribute__ ((at(addr)))
typedef unsigned char __sfr_byte;
typedef struct {                          // 定義結構體
   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 {                          // 定義聯合體
   __pa_bits bits;                       // 結構體的使用
   __sfr_byte byte;
}  __pa_type;
DEFINE_SFR(__pa_type, __pa, 0x1a);       // 定義 PA 寄存器
#define _pa         __pa.byte            // 聯合體成員的使用
#define _pa5        __pa.bits.__pa5      // 結構體成員的使用
```

枚舉類型定義時如果沒有指定初值，則其值由上一個有賦值的開始，依次加 1。如果全部沒有指定初值，則由 0 開始，依次加 1。

代碼清單 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 結構體與聯合體的區別

結構體與聯合體的最主要區別是空間的分配，結構體會為各成員分配內存空間，而聯合體內各成員共用一個內存空間，以佔用內存空間最大的成員為聯合體分配內存空間。

如例子 8 中，如果使用 sizeof(__pa_type) 求聯合體佔用的空間得到結果是 1，如果把 _pa5 置 1，則 __pa_type 內的 byte 成員的第 5 位也會被置 1。

1.5 預處理，巨集定義與內聯函式的區別

標準 C 語言中預處理有 3 種：巨集定義 (#define)、檔包含 (#include)、條件編譯 (#if 等)，預處理是在編譯之前執行的，如巨集展開的時機是在預編譯時進行。

1、巨集定義：註意使用巨集定義時，只是簡單的把巨集名替換為巨集定義的內容，並不做任何運算，如 #define S(r) 3.1415926*r*r，如果使用 S(6+6)，結果就成了 3.1415926*6+6*6+6，而不是最初想要的結果，可以使用 #define S(r) 3.1415926*(r)*(r) 得到想要的結果。

2、檔包含：使用 #include 可以把相關檔包含進來，包含檔時，使用雙引號和使用“<>”不同，一般雙引號用於用戶自定義的檔，“<>”用於庫文件，以減少在編譯時因搜索路徑所花費的時間，註意頭檔的重覆包含問題。

3、條件編譯：選擇源代碼中的一支來編譯，而不用把全部源代碼都進行編譯，常用在調試和在不同機器上移植代碼時使用，有 3 種形式 (else 分支可以沒有)：

(1)、#ifdef 條件

程式段 1

#else

程式段 2

#endif

(2)、#ifndef 條件

程式段 1

#else

程式段 2

#endif

(3)、#if 條件

程式段 1

#else

程式段 2

#endif

4、巨集定義與內聯函式的區別：

(1)、巨集定義只做簡單的替換，替換時不做任何運算。

(2)、巨集定義在預編譯時進行，內聯函式則是在有函式調用時進行。

(3)、巨集定義的參數不能指定數據類型，內聯函式則必需要指定。

(4)、編譯器對巨集定義本身的內容不做任何檢查。如使用 #define error **8，不會報錯，而內聯函式則不行。

例子 9：預處理綜合實例

代碼清單 1.16：

```
FUNCTION.H
#ifndef _FUNCTION_H              // 如果去掉 #ifndef，則報重定義出錯
#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                              // 調試時用

u8 _debugval;

void main()
{
   u8 ch;
   const u8 *Led_String = "YOU";
   while(*Led_String){
          #if DEBUG                          // 如果把 1 改為 0 則不編譯該語句
          _debugval = *Led_String;
          #endif
          ch = *Led_String;
          ch = LeftShift(ch, 1);
          GetKey(ch, 0);
          Led_String++;
   }
}
```

如果向巨集定義 Max(num1, num2) 中的 num1 傳入 ch++，mun2 傳入 0，則 (ch++) > (0) ? (ch++) : (0)，此時 ch 的值就與預想的不一樣，所以使用巨集定義時要多註意。

1.6 流程控制

1.6.1 三種執行流程

C 語言的執行流程有：順序執行、選擇執行、循環執行。

1.6.2 判斷語句 if、switch 的使用

if 和 switch 都具有判斷的作用，當 if 的條件情況太多時一般使用 switch 替換，switch 的條件的類型只可以是除浮點型之外的基本數據類型和枚舉型。case 只能帶常數 (不包括 const 申請的常數)，每個 case 執行後要使用 break，否則會繼續執行下面的語句，多個 case 可以執行同一些語句，default 是默認的情況，default 可以不加 break。

代碼清單 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 循環與循環的嵌套

1、while(表達式) 語句

2、do…while(表達式);

兩者的區別：do…while(表達式)；至少會執行循環體內的語句一次。

代碼清單 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);`
	`}`

執行結果：sum = 0	sum = 11

3、循環的嵌套

在循環體中可以再執行循環體。

1.6.4 break 與 continue

break 只能用於循環和 case 語句，continue 只能用於循環語句，兩者的區別是：在循環中如果遇到 continue 則執行下一次該循環體的語句，break 則是直接跳出了本層循環體，執行循環體外的語句。

代碼清單 1.19：

```
while(1)
{
   int j = 0;
   while(1)
   {
      j++;
      if(j == 5)
      {
        continue;          // 執行本層循環的下一次循環
      }
      if(j == 10)
      {
        break;             // 跳出本層循環
      }
   }
}
```

1.6.5 正確使用 goto

goto 可跳到本函式內任何一個標號處執行語句，一般不建議使用。

例子 10：正確使用 goto 可以優化代碼之場景

代碼清單 1.20：

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

```
BOOL fun1()
{
   ……
}
```

```
BOOL fun2()
{
    ……
}
```

```
BOOL fun3()
{
    ……
}
```

本來版本：

```
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;
       return;
   }while(0);
   result = 0x55;
```

```
    p = b;
}
```

1.7 作用域

無論是變數還是函式，都具有其作用域。全域變數 / 函式在整個工程中使用 extern 後就可以使用，如果在全域變數加 static 則只能在本檔內使用，為了節省 ROM 空間，函式一般不使用 static，局部變數在函式內申請之後的語句都可以使用，如果是在循環，if，switch 內申請的變數，則執行完這些語句之後，下面的語句就不能再使用這些變數，如代碼清單 1.19 中的 j 變數，出了外層 while 循環後的語句將不能夠再使用。

第二章 C compiler V3 擴展及其限制

2.1 在 HT-IDE3000 中設置 C compiler V3

2.1.1 使用環境

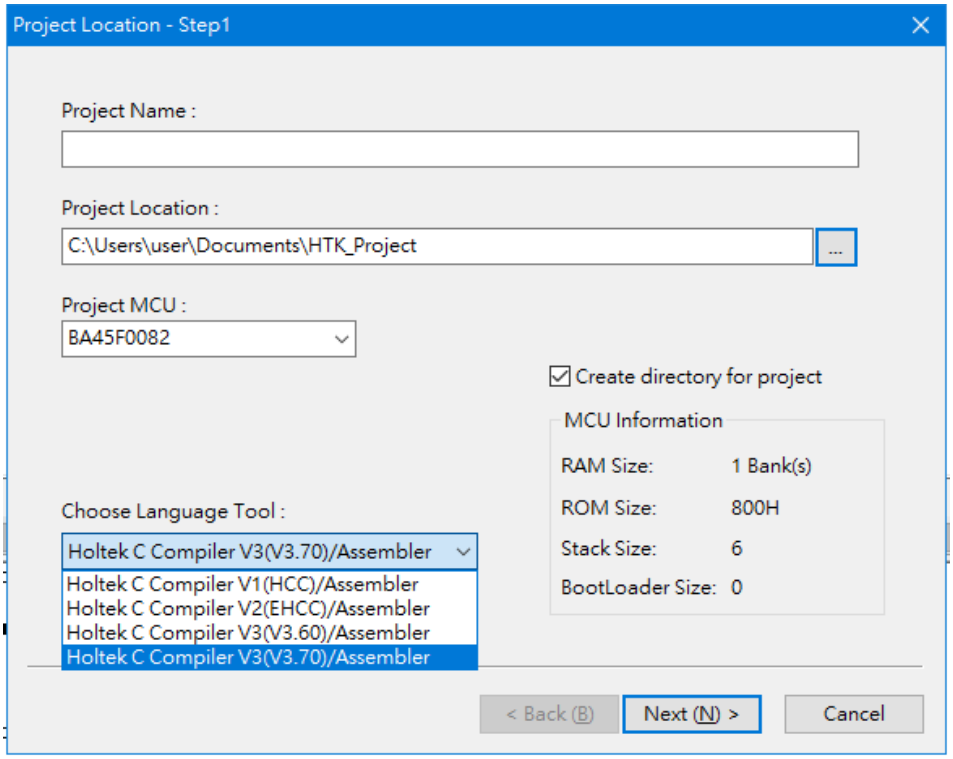
本文於 HT-IDE3000 V8.1.6 的基礎上編寫。

註：HT-IDE3000 V7.71 開始支援 C compiler V3

2.1.2 新建項目時，選定 C compiler V3 編譯器

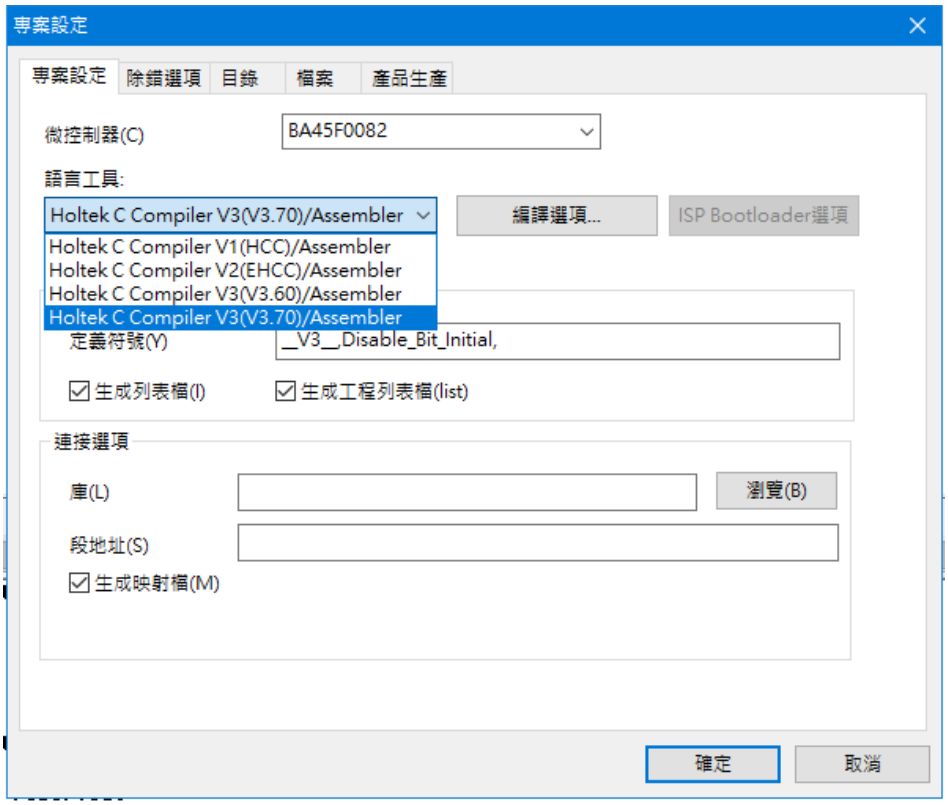
進入 HT-IDE3000 開發環境後，依照下列方法建立一個新的項 (project)：

- 移動滑鼠遊標到 Project 選單，按左鍵。
- 移動滑鼠遊標到 New 命令，按左鍵。
- 出現如下的視窗，在 Choose Language Tool 之處勾選 Holtek C Compiler V3 (V3.xx)/Assembler。



2.1.3 開啟專案後，如何選用 C compiler V3 編譯器

若項目 (project) 已開啟之後，可以點選 Option 選單下的 Project Setting 命令，在 Choose Language Tool 中點選 Holtek C Compiler V3(V3.xx)/Assembler 以設定使用 C compiler V3 版編譯器。



2.1.4 專案編譯選項設定

1、定義符號

如下圖紅框設置，IDE 將會給編譯器傳參數：-DGCC_COMPILER -DCOUNT=5

相當於巨集定義：

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

且其有效範圍為整個 project，多個巨集定義用‘,’隔開。

使用巨集的一個例子是條件編譯：

E.G.1:

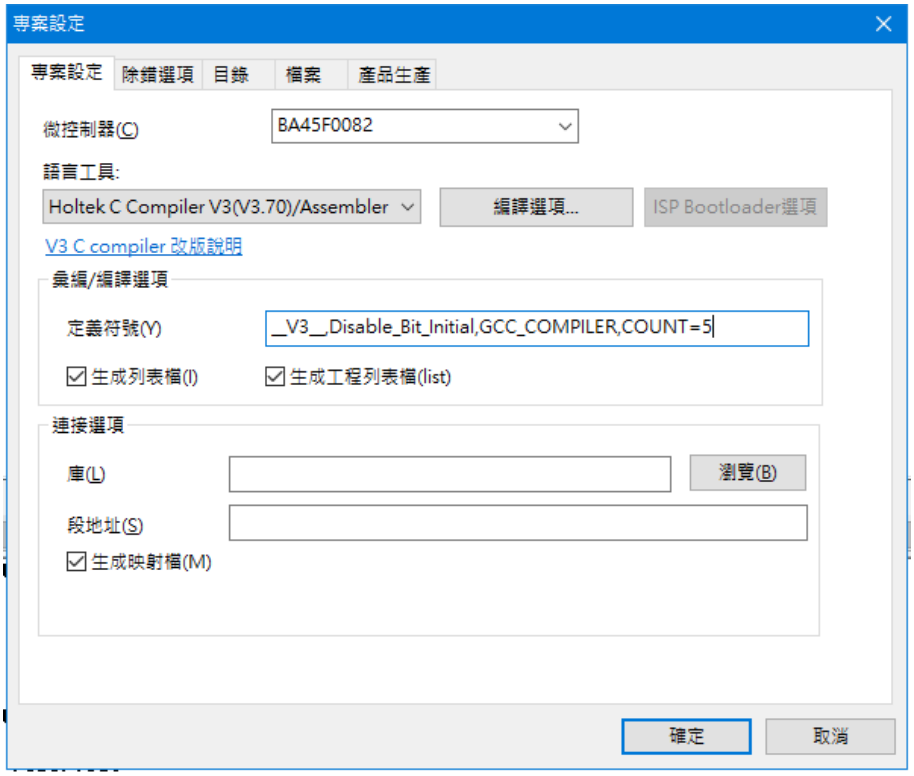
```
#if GCC_COMPILER
typedef unsigned int uint16;
#else
typedef unsigned int uint8;
#endif
```

E.G.2:

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

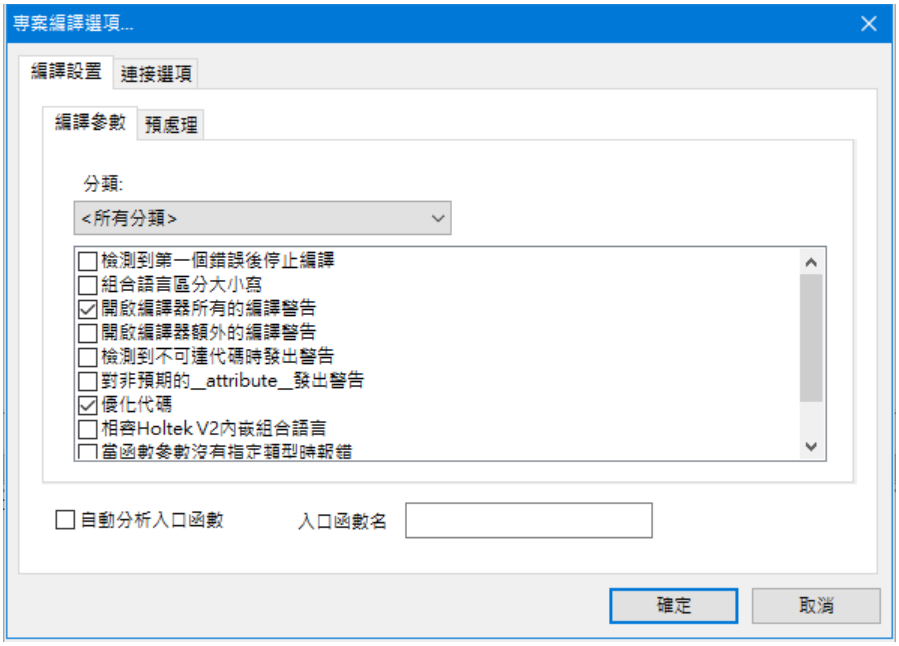
```
x = 6;
#else
x = 7;
#endif
```

註意：使用 C Compiler V3，IDE 會默認為其傳巨集定義
“__V3__,Disable_Bit_Initial”



2、編譯參數

使用 C Compiler V3，可以選擇編譯選項，IDE 默認會傳優化參數 -Os，點擊專案設定下的“專案編譯選項”，彈出對話框如下：



組合語言區分大小寫：用於混合語言工程，若一個工程既有 c file 又有 asm file 時，可以開啟這個選項 (C 語言區分大小寫，asm 默認不區分)。

組合語言字元串，2 個字元佔用一個 word：用於 asm file，當所選 MCU 寬度為 16 bits (如 HT66F50) 時，可以開啟這個選項，則字串中的每兩個字元會佔用一個 word，否則默認一個字元佔用一個 word。若所選 MCU 為 14, 15bits (如 HT46R4AE)，則不可啟用這個選項。

開啟編譯器額外的編譯警告：比如：缺省的函式返回值，無符號數與 0 的比較等等。

優化代碼：優化選項。

相容 Holtek C V2 內嵌組合語言：可以使用 #asm/#endasm 的語法完成內嵌組合語言功能

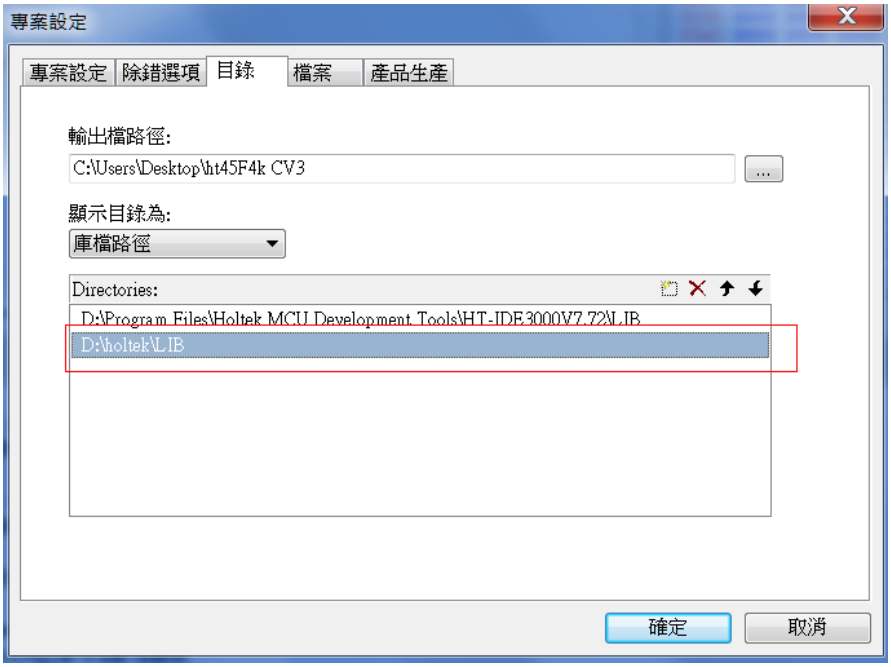
存取 const 變數時使用查表指令進行優化：有 TBHP 寄存器且 PROM 寬度為 16bit 的無擴展指令 MCU 會有此選項，勾選之後 table 的大小減半，若要指定 table 的位置，此選項必須勾選。

配置 enum 類型為 byte：enum 編譯器默認為 int，當定義的 enum 數值不大於 127 時，勾選此選項，可以節省空間。

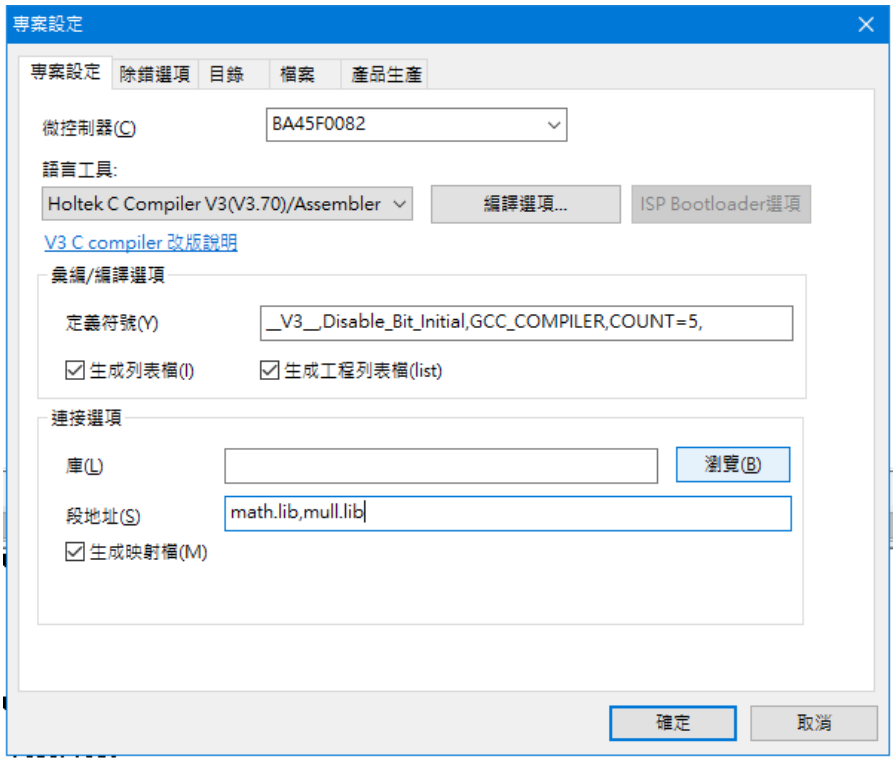
2.1.5 連接選項

2.1.5.1 增加庫檔

1、先將庫文件路徑目錄加入目錄，如下



2、敲入庫檔案名稱，如下 math.lib, mull.lib 用 ‘,’ 隔開。



2.1.5.2 設定函式位址

格式：_fun_name=addr, _fun2_name=addr2, …

其中，_fun_name 為 ‘_’ + 函式名稱，addr 為所要配置的位址，為 16 進制。多個函式用 ‘,’ 隔開。

註：不建議使用此功能，會影響編譯器的優化。

2.1.5.3 生成映射檔

勾選後將生成 .map 檔。

2.1.5.4 連接選項

優化 RAM 空間 (不允許巢狀中斷)：優化 RAM 空間，不允許在一個中斷裏面進入另一個中斷。

刪除沒有被調用的函式 (針對 C)：若一個函式不被調用，將不為其分配空間，這個選項默認開啟。

未指定初值的全域變數 / 靜態變數其值默認為 0 (不含 bit 類型變數)：若 global 變數沒有初始值，則默認其初始值為 0。在程式開始運行的時候，將其初始化為 0，此操作不包含 bit 變數。

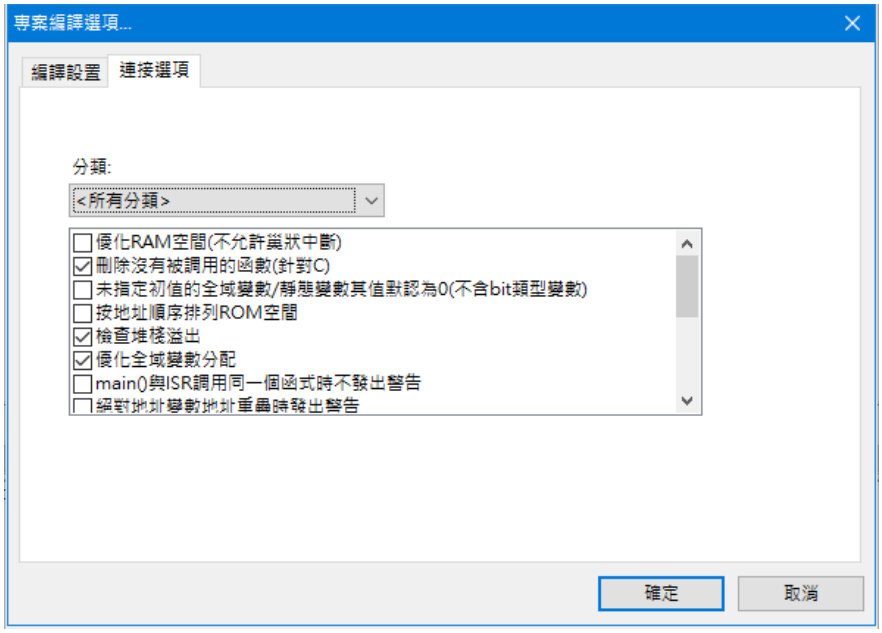
按地址順序排列 ROM 空間：對於多 ROM bank 的 MCU，為節省 BP 切換指令，編譯器不一定會按 ROM bank 順序分配程式，如果希望按 bank 順序排列，可以勾選此選項。

檢查堆棧溢出：當主程式的堆疊調用層數超過此 MCU 的 stack 數量時，發出警告。

優化全域變數分配：對具擴展指令 MCU，將調用次數多的全域變數優先分配於 RAM bank 0。

main() 與 ISR 調用同一個函式時不發出警告：編譯器不允許主程式與中斷調用同一個子函式，但如果使用者確定此用法沒有問題 , 可勾選此選項。

絕對地址變數地址重疊時發出警告：連接器會檢查絕對位址變數使用的位址有重疊時發出警告，但如果位址完全重疊，會被認為是同一變數，此時沒有警告。



2.2 C compiler V3 擴展語法及功能

2.2.1 中斷服務程式

若微控制器的周邊裝置具有中斷功能，程式也需要此中斷機能以完成工作時，則必須定義此周邊裝置的中斷服務函式 (Interrupt Service Routine，ISR)，如下的格式：

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

中斷服務函式必須遵守下列規定：

→ 返回的資料類型必須是 void

→ 不能有參數

→ 必須使用 __attribute__((interrupt(0x0c))) 設定中斷向量值 (interrupt vector)

→ 中斷入口會自動保存寄存器 (ACC, BP, STATUS, MP, TBLP, TBHP)，並在中斷出口時恢復。

→ 既可以被中斷服務程式訪問也可以被其他函式訪問的全域變數應定義為 volatile。

註：MP/TBLP 只在中斷函式有使用到時才會保存。

中斷服務函式的使用註意事項：

C Compiler V3 支援中斷內部調用函式，但不同中斷與 main 之間不能調用同一個函式，會造成 RAM 重疊，對此現象 linker 將偵測出並報 warning (若被調用的函式無宣告及使用任何的 local 變數可忽略此 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 與 isr1 都調用了 fun1，即為共同調用。
- 雖然 isr2 沒有直接調用 fun1，但它調用了 fun2，fun2 又調用 fun1，所以它與 main，isr1 也造成共同調用，調用圖可以看 map 文件。
- 同樣，各 ISR 之間，也不能調用同一個函式，除非能夠保證在中斷 1 執行過程中不會進入中斷 2。

2.2.2 絕對位址變數 (absolute variable)

將一個變數指定到固定的位址，比如 _bp 在 [04H]，可以寫成如下：

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

編譯器會將此變數分配到這個位址，但如果這個變數在整個程式中都沒有使用到，那麼連接器 (linker) 有可能將其它變數分配到這個位址上，編譯器會將之翻

成組合語言的 EQU 指令，如下：

```
__bp EQU 4h
```

在頭文檔 ht48c70-1.h 中，有定義它的簡寫：

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

說明：使用 volatile 關鍵字修飾，以防被優化。

所以如果程式有 include HTxx.h 也可寫成：

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

陣列、指標、結構體等變數也可以定義成絕對變數，它們的定義方式與一般變數一樣，只是多了個地址：

```
static volatile unsigned char arr[10] __attribute__ ((at(0x140)));
// 陣列
static unsigned int *volatile p __attribute__ ((at(0x040)));
// 指標
typedef struct
{
   int a;
   int b;
}my_type;
static volatile my_type ab __attribute__((at(0x040))); // 結構體
```

2.2.3 MCU 頭文檔介紹

HT-IDE3000 有自帶 MCU 的頭文檔，裏面主要是一些寄存器及標誌位的定義。

在工程中，只需要寫 #include “MCU.h”，比如：#include “ht66f60.h”就會自動引入頭文檔。

頭文檔的主要內容：

1、絕對位址及中斷語法的簡寫：

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

其中，isr_name 表示中斷服務程式的名字，vector 表示中斷的地址，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 表示特殊寄存器是靜態的，因為每個 C file 都有可能 include MCU 頭文檔，所以特殊寄存器必須定義成 static, sfr_type 表示它的類型，sfr 為特殊寄存器名稱，addr 表示它所在的地址。

比如：DEFINE_SFR(unsigned, _bp, 0x03)

2、特殊寄存器的定義：

```
#define _ACC      __ACC
```

其中 __ACC 已經是用 DEFINE_SFR 定義好的變數，用戶可以直接在程式中使用 _ACC

3、標誌位的定義：

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

其中 __status 是已經定義好的變數，如果 user 想要自定義標誌位，可以參考它的定義方式，如下：

a. 需先定義一個 struct (STATUS 寄存器的所有標誌位的集合) 及 union (使用 STATUS 可以整個 byte 操作，也可以單獨操作它每一個 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 的位址在 0x0a，如果是一般的 bit 變數則不需要指定位址

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

c. 再將 _pa0 巨集定義為其中的一個位域，方便引用。

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

4、內建函式的定義：

內建函式	作用
GCC_RL(varname)[1]	varname 不帶進位左移
GCC_RLC(varname) [1]	varname 帶進位左移
GCC_RR(varname) [1]	varname 不帶進位右移
GCC_RRC(varname) [1]	varname 帶進位右移
GCC_NOP()/_nop()	執行一個空操作 (NOP)
GCC_SWAP(varname) [1]	varname 的前四位與後四位交換
GCC_HALT()/_halt()	執行一個 HALT 指令
GCC_CLRWDT()/_clrwdt()	清零看門狗計時器 (CLRWDT)
GCC_CLRWDT2()/_clrwdt2()	清零看門狗計時器 (CLRWDT2)
GCC_CLRWDT1()/_clrwdt1()	清零看門狗計時器 (CLRWDT1)
GCC_DELAY(n) [2]	延遲 n 個指令週期

註：[1] varname 必須是一个 8bit 變數
[2] 0 <= n < 263690

5、若 MCU 帶 EEPROM，則定義 __EEPROM_DATA(a, b, c, d, e, f, g, h);
可以直接使用此語法對 EEPROM 寫值：

```
#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));
```

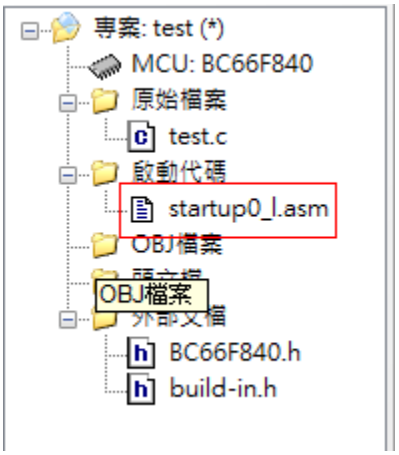
一次可以按順序寫 8 個值，比如：

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

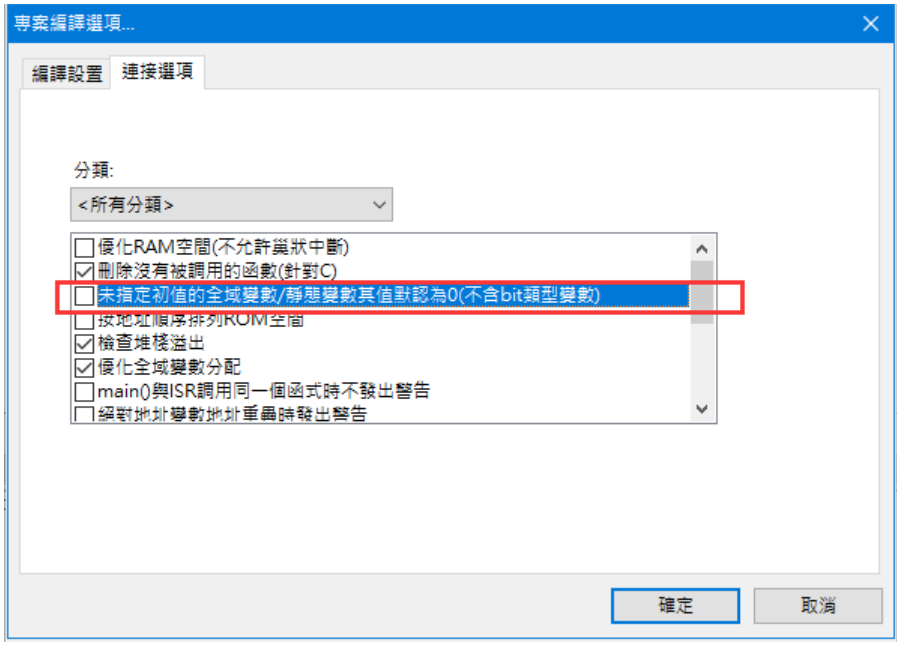
註意，__EEPROM_DATA 不能寫在函式體內。

2.2.4 變數初始化

C Compiler V3 支援變數的初始化，並在程式一開始調用一個 startup 的程式來實現，此 startup 程式有兩個文檔，startup0 及 startup1：



1、一般建立工程時，未初始化的全局變數默認不初始化為 0，如果有需要再加勾選項，如下，勾選選項即可 (此功能在 IDE7.8 及以上版本實現)



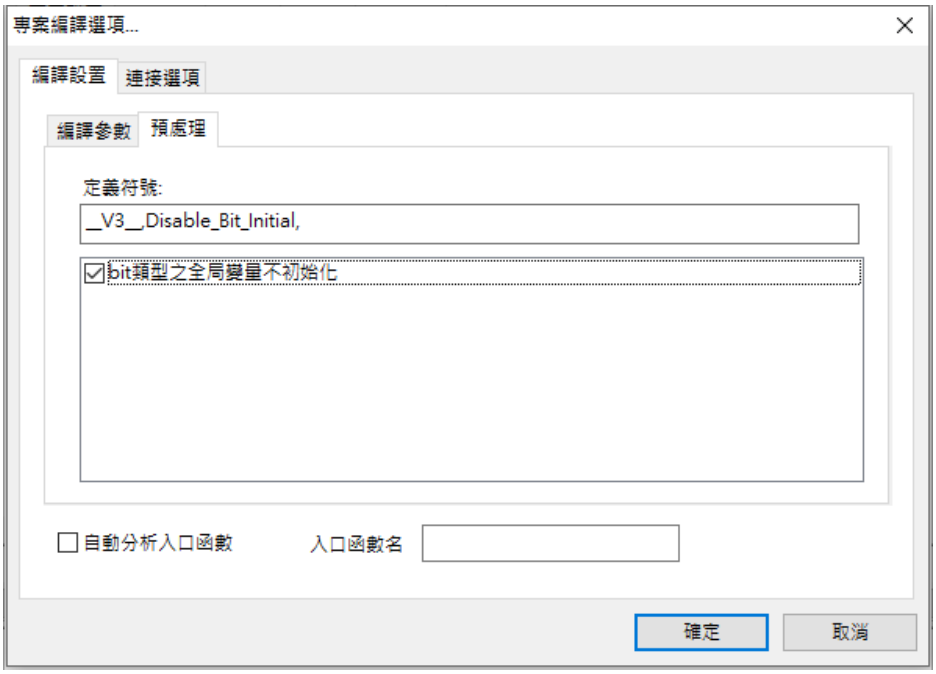
2、Startup 程式的代碼是開放的 (此功能在 IDE7.72 以上版本實現)

3、更新 IDE 版本後，需將工程文件夾下的 startup 文件刪除，讓 IDE3000 重新載入

4、編譯器預設不對全域 / 靜態 bit 變數做初始化，如果要修改此設定，可以在此設置 (IDE3000 8.05 及以上版本)：

a. 勾選選項，不管是否有設初始值，編譯器都不對 bit 變數初始化。

b. 不勾選選項，編譯器對 bit 變數初始化，若無初始值，將默認為 0。



2.2.5 內嵌彙編語言

如果想要讓編譯後的程式碼更為精簡，執行上更有效，可以在 C 程式中加入組合語言的指令。語法：

1、asm (“opcode [operands]”);

compiler 直接輸出指令 opcode [operands]

e.g.

```
extern void FUN() ;
asm("extern _FUN_PAR1:byte");        // 外部函式參數的聲明
void main( )
{
   asm("mov a,1");
   asm("mov _FUN_PAR1,a");
   asm("call _FUN");                 // 函式調用
}
```

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

其中 varname 為變數名，為防止被優化，可加 volatile 修飾。

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));     // 局部變數讀值
   volatile asm("mov %0,a" : "=m"(i) : "m"(i));      // 全域變數寫值
   while(1);
}
```

3、#asm/#endasm

當選擇“相容 Holtek C V2 內嵌組合語言”編譯選項時，可以使用以下語法，輸入一整段內嵌彙編

e.g.

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

註意：

- 每條語句只能寫一條指令，需用引號。
- 第二種內嵌彙編格式只能出現在函式內，且變數大小不能超過 1byte。
- 第一種內嵌彙編格式可以出現在函式外，也可以用來定義變數，section 等，其輸出內容為引號之間的字串，編譯器不做處理。

2.2.6 指定函式的位址

C Compiler V3.20 以上版本支援將函式指定位址，語法：

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

關鍵字 __attribute__((at(addr))) 將 function 地址定義在 addr，比如：

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

表示把地址定義在 0x373，指定位址的函式支援參數及返回值。

2.2.7 指定 const 的位址

對 ROM 寬度有 16bit 且帶 TBHP 寄存器的 MCU，C Compiler V3.20 以上版本支援將 const 指定位址，語法：

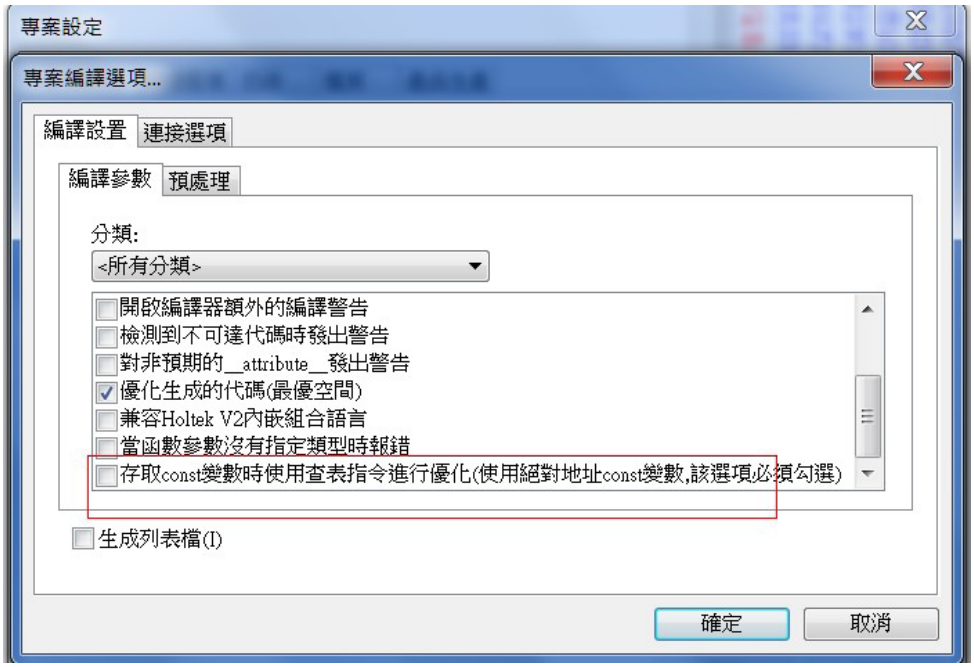
```
const char __attribute__ ((at(addr)))  cvar1 [3]={1,2,3};
```

關鍵字 __attribute__((at(addr))) 將地址定義在 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))) 表示把地址定義在 0x123。

說明：使用該功能需要勾選如下圖所示之選項



2.2.8 變數分配

Holtek MCU 有兩個 space，ROM 和 RAM。變數的分配規則如下：

1、一般變數會佔用 RAM 空間，其初始值存放在 ROM 空間

2、const 變數分配 ROM 空間，但如果使用 volatile 修飾且無指定地址，則佔用 RAM 空間

3、如果靜態變數在程式中沒有被改變值，且傳優化參數，則編譯器當其為 const，佔用 ROM 空間

2.2.9 __attribute__ 關鍵字

__attribute__ 是 C Compiler V3 的特有關鍵字，目前，V3 支援的 __attribute__ 的用法如下：

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

指定一個函式的屬性為入口函式

語法：

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

說明：

a. entry 無參數

b. 返回值類型為 void

c. 參數類型為 void

d. 不限制 main/isr 設定 entry，但此時 attribute entry 將無效。

範例：

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

2、__attribute__((at(addr)))

指定一個函式 / 變數的地址

語法：

指定函式：

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

指定變數：

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

說明：

a. addr 為指定地址，不可缺失

b. 變數定義了 __attribute__((at(addr)) 屬性後需定義成 static

c. const 變數也可定義 __attribute__((at(addr)) 屬性，僅限於具擴展指令 MCU 或具有 TBHP，ROM 寬度為 16bit 的 MCU，此時 addr 為變數在 PROM 中的地址。

d. main/isr 也可設定 __attribute__((at(addr)) 屬性

範例：

```
__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)))

定義一個中斷向量入口地址

語法：

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

說明：

addr 是中斷入口地址，不可缺省，而且必須為 4 的倍數

範例：

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

一個函式可以定義多種屬性：

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

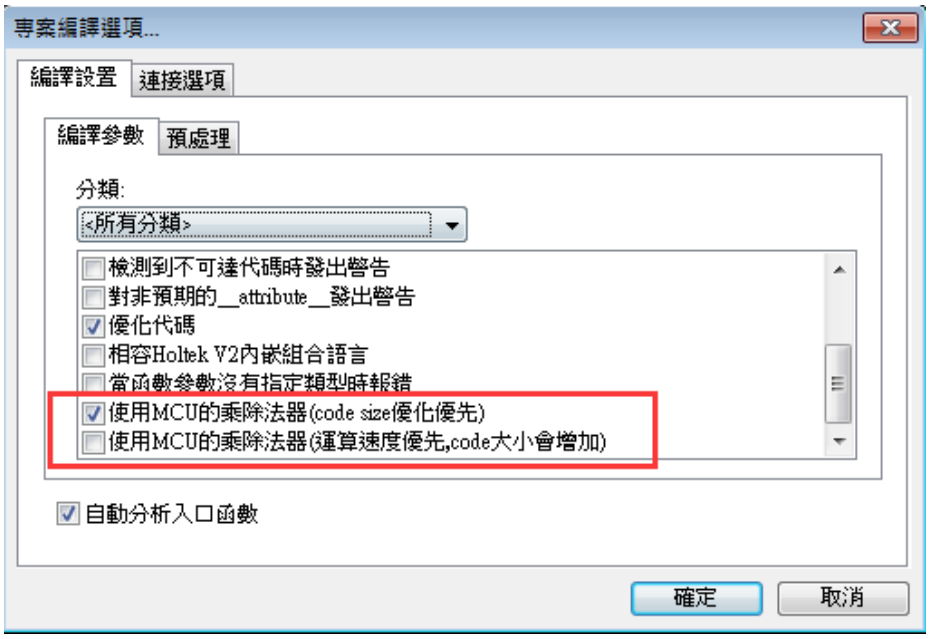
b. `__attribute__((interrupt(addr),at(addr)))`

2.2.10 硬件乘除法器

1、硬件乘除法器 (MDU) 功能適用于具有 MDU 的 MCU，可查看 datasheet，若有 MDUWR0、MDUWR1、MDU0R0…暫存器，說明此 MCU 有 MDU 功能。

2、IDE3000 V7.90 支援 MDU 功能。

3、對具有 MDU 功能的 MCU，IDE3000 下“選項 → 專案設定 → 編譯選項 → 編譯參數”會出現以下兩個參數：



A. “使用 MCU 的乘除法器 (code size 優化優先)”：表示啟用 MDU 功能，為默認勾選選項，需注意，若從舊版 IDE3000 (V7.89 及更早) 移植過來的程式，需確定此選項有無勾選。

B. “使用 MCU 的乘除法器 (運算速度優先，code 大小會增加)”：因大部份 MCU 的 MDU 都為 16bit，所以對于 char/unsigned char 類型的乘除法運算，若使用 MDU 功能，使用的 code size 不一定會更少，但是運算速度會更快，所以此選項適用于追求速度優先的用戶。

4、大部份 MCU 的硬件乘除法器只有 16bit×16bit、32bit/16bit、16bit/16bit 三種 (HT66FM5440 有 8bit×8bit 及 8bit/8bit)，所以并不是所有的乘除法運算都會使用硬件乘除法器：

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 的相關運算要選擇參數“速度優先”才能使用 MDU。
b. 除法運算需等式左邊類型小于 Long 才能使用 MDU。

c. 除法部份，左行是被除數。
d. U 表示 unsigned。

5、使用硬件乘除法器與乘除法運算庫的效益對比：

	乘除法運算庫		MDU	
	Size (word)	執行時間 (最壞情況下使用的指令週期)	Size (word)	執行時間
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 數據類型

1、C Compiler V3.5 以上版本 (HT-IDE3000 V7.93 以上) 開始支援 bit 數據類型。
2、bit 變數佔用一個 bit, 最低位有效。
3、bit 變數不支援 stuct/union/const/register/ 陣列 / 指標 / 函數參數及返回值。
4、可以指定 bit 變數的位元址，語法如下 ：

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

其中，addr 表示其所在位址單元，val 表示第幾位。
比如：

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

表示 flag1 將佔用 [80h].3 的位址。
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 函式指標

C Compiler V3 不支援函式指標，比如如下的程式會報 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 遞迴函式

Holtek MCU 並不支援遞迴呼叫，但一種比較特殊的遞迴呼叫 (尾遞迴呼叫)，編譯器可以做優化將它轉化成非遞迴函式，詳見 3.10 節所述。

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

部分舊架構的 Holtek MCU 的 MP 只有 7bit，為達到更好的優化效能 C Compiler V3 不支援此類 MCU，具體的 MCU 型號可參考文件 < Holtek C Compiler V3 FAQ >。

2.4 編譯器管理的資源

Holtek MCU 的某些特殊功能寄存器由 C Compiler V3 使用，用戶使用時要小心，下表列出這些寄存器，以及它們對編譯器的主要用途，進入中斷服務程式時會自動保存所有用到的寄存器。

編譯器用到的特殊寄存器	主要用途
MP/IAR	用于存取 RAM space 的值，搭配 RAM BP 使用
BP	RAM BP 用於存取 RAM space 的值，ROM BP 用於訪問 ROM space
STATUS	用于表達式計算
TBLP	用于 table read 或存儲 RAM BP
TBHP	用于 table read
TBLH	用于 table read
ACC	存儲函式返回值等
PCL	用于 table read

第三章 C compiler V3 的優化功能

3.1 優化內容介紹

C Compiler V3 是一種優化編譯器，其進行優化的主要目的是簡化代碼，減少代碼量。在預設情況下，啟用 -Os 將執行所有的優化功能。下表概括了編譯器執行的每項優化功能，包括每個優化功能是否會影響調試。

	是否影響調試	節省 ROM	節省 RAM	節省 Stack
代數轉換 (Algebraic Transformations)	√	●		
複製傳遞 (copy propagation/value propagation)	√	●		
刪除執行不到的代碼 (Unreachable code Elimination)	√	●		
刪除死代碼 (Dead-code Elimination)	√	●		
常數折疊 (Constant Folding)		●		
常數傳播 (constant propagation)		●		
內聯程式 (Inline Procedure)	√			●
強度削減 (Strength Reduction)	√	●		
尾遞迴呼叫 (Tail Recursive Call)				●
子表達式刪除 (subexpression elimination)		●		
尾部合併 (Tail merging)	√	●		
Bp 優化		●		
Dead section 刪除	√	●		

3.2 代數轉換 (Algebraic Transformations)

代數轉換指的是，compiler 會將表達式的某些運行用更為簡潔的相同功能的運算代替，比如：

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

這種替換不會影響到調試。

3.3 複製傳遞 (copy propagation/value propagation)

複製傳遞是這樣一種變換，對於變數 x 和 y，進行 x ← y 賦值後，那麼在後面的代碼中，只要中間插入的指令沒有改變 x 和 y 的值，可用 y 代替 x 。這種優化本身並不能節省指令，但是能實現刪除死代碼 (請參閱第 2.3.5 節“刪除死代碼”)。例如下面的 C 原始程式碼：

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

說明：05 行可以轉化為 c=a; 這樣 04 行就成為無用代碼，可以刪去，連同變數 b 也可以刪去。

複製傳遞，會影響到用戶的調試，因為 04 行及變數 b 被刪去，那麼，user 將不能在 04 行設置斷點，在變數監視 (watch window) 裏面也看不到變數 b。

3.4 刪除執行不到的代碼 (Unreachable code Elimination)

刪除執行不到的代碼優化功能刪除在正常的程式流程中不會執行的代碼。例如下麵的 C 原始程式碼：

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

顯然，上面代碼片段中的 else 部分絕對不可能執行到。使用此優化後，生成的彙編代碼將不包含 else 部分的指令，刪除執行不到的代碼優化可能對在 C 原始程式碼的某些行設置中斷點產生影響。

3.5 刪除死代碼 (Dead-code Elimination)

在函式中計算，但在至函式出口的任何路徑上都沒有使用過的值視為“死”值。只計算死值的指令視為“死”指令。函式作用域外的值視為使用過 (不是死值)，因為不能確定這種值是否使用過。可以參數第 2.3.3 節“複製傳遞”中的例子。

刪除死代碼優化可能會對在 C 原始程式碼的某些行設置中斷點及變數監視產生影響。

3.6 常數折疊 (Constant Folding)

常數折疊是指 compiler 會對表達式中一些可以內部計算的值直接計算，不為其生成彙編代碼：

Example:

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

compiler 將輸出如下語句的彙編 →

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

常數折疊不會影響到調試。

3.7 常數傳播 (constant propagation)

一些表達式，雖然裏面的變數不是常數，但 compiler 可以通過簡單的計算算出它的值，比如：

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

compiler 將輸出如下語句的彙編 →

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

常數傳播不會影響到調試。

3.8 內聯程式 (Inline Procedure)

一個子函式比較簡單時 (如下條件)，調用它的函式會在調用時直接將它展開，從而減少堆棧的使用：

內聯函式的一些規定：

1、函式不能包含複雜的語句，這些複雜的語句包括：switch 語句、for 語句、while 和 do while 語句、goto 語句。

2、函式不能是遞迴函式。

3、函式體內的函式語句不能超過 30 行。

4、子函式與其調用函式在同一個 C file。

Example:

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

內聯之後的函式：

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

內聯函式的弊端：

代碼量的膨脹，函式每展開一次，它的代碼量便增加一次。

如果想要避開內聯函式，可以將子函式與母函式使用不同的文檔編寫。

內聯函式的優勢：

1、函式展開後，可能有其它的優化可以進行

2、減少堆疊的使用

3、子函式被展開後，若沒有被其它函式調用，則會被 Linker 當成 dead section 刪去 (2.3.14 節所述)

內聯函式會使得被展開的子函式無法設斷點。

3.9 強度削減 (Strength Reduction)

強度削減指的是，一些簡單迴圈有可能會被化簡為一般的語句，比如：

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

削減成：

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

強度削減會使得被化簡的代碼無法設斷點。

註意：因為編譯器只關心執行的結果，所以，通常有特殊目的的代碼 (比如通過迴圈來達到 delay 功能) 會因此而失去意義，建議若有此類的代碼，可以將迴圈體用內嵌彙編實現，或者將變數用 volatile 定義。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 尾遞歸調用 (Tail Recursive Call)

Holtek MCU 並不支援遞迴呼叫，但一種比較特殊的遞迴呼叫，編譯器可以直接幫助我們做優化，將遞迴呼叫直接優化為非遞迴呼叫，這一類遞迴呼叫在編譯器裡面稱為尾遞迴呼叫 (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);
}
```

像上面的一個代碼，只有在代碼的最後一行調用函式自身的代碼稱為尾遞迴，編譯器將遞迴優化成一個迴圈：

```
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;
}
```

尾遞歸調用不會影響 debug。

3.11 子表達式刪除 (subexpression elimination)

子表達式刪除是指在幾個表達式中，若有部份的子表達式是相同的，那麼編譯器會先計算這部份，儲存在一個臨時變數裡，之後直接使用此臨時變數運算，比如，如下 b*c 為公共的子表達式，可以先計算在 tmp 裡，這樣可以減少一次 b*c 的計算。

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

刪除子表達式不會影響 User 的 debug。

3.12 尾部合併 (Tail merging)

尾部合併優化將多個相同的指令序列合併為一個指令序列。例如下面的 C 原始程式碼片段：

```
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 與 ln08 是同一段代碼，這時編譯器只會編譯 ln08 而刪去 ln03，這時如果

執行的是 if 分支，則執行完 ln02 時，就會跳到 ln08，有可能會誤導 user，以為跑到了 else 分支，但其實不是。

因為兩行或更多行原始程式碼可能共用同一彙編代碼序列，這使調試器很難確定正在執行哪一行原始程式碼。

3.13 ROM BP 優化

對多個 ROM bank 的 MCU，使用 Jmp 或 call 之前，需設定 ROM BP 來決定是哪一個 bank，如果當前 BP 與 jmp/call 之後的 BP 相同，linker 會刪除它多餘的 mov BP, a 指令。

若 ROM BP 指令不能刪除，則會使用相應的 set/clr BP.5(,6,7) 指令，減少 ACC 的使用。比如：

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

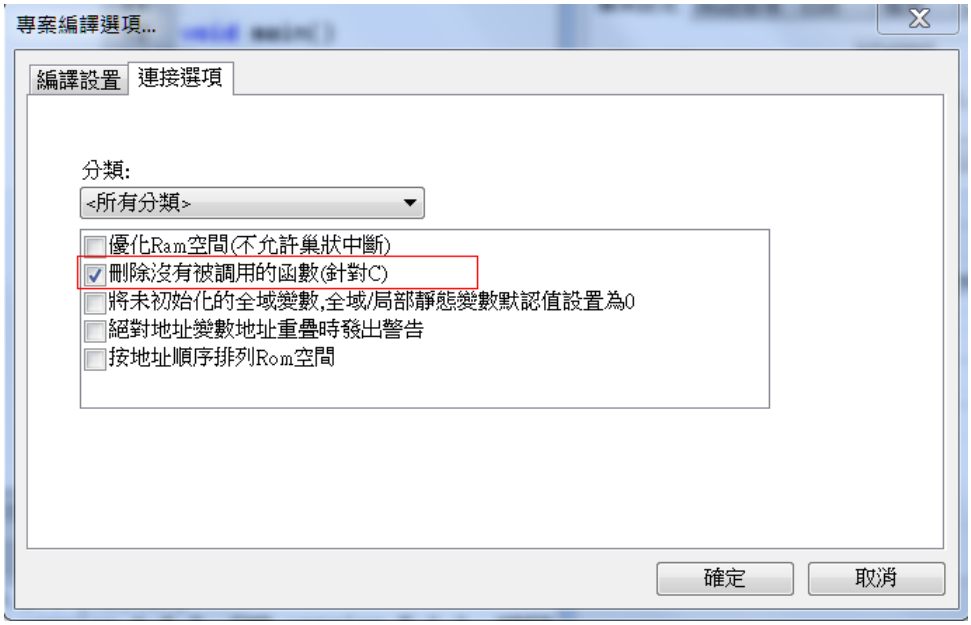
若 fun1 與 fun2 被分配到同一個 bank，則會翻譯成 call _fun2

若 fun1 被分配到 bank0，而 fun2 被分配到 bank1，則會翻譯成：

```
set bp.5
call _fun2
clr bp.5
```

3.14 Dead section 刪除

若一個 function 在整個程式中都不被呼叫，則 linker 不為它分配空間，此功能可以通過如下的選項選擇，特別是，當 compiler 將函式呼叫優化成 inline 展開時，則子函式有可能不被調用而被刪除。若此工程含有 ASM 檔，則此功能無效。



第四章 Holtek C V1，V2，V3 及 ANSI C 的差異對比

4.1 數據類型

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 位浮點數皆使用 IEEE754 32 位格式

Holtek C V3 24 位浮點數格式：

只有 4~5 位精度 (V3.20 以上版本支援)

符號 sign	指數 e	尾數 m
23	22~15	14~8　7~0

bit 形態不可用於指標 (pointer) 的數據形態，不可定義為 const。

4.2 陣列

維數	V1 (最大陣列長度)	V2 (最大陣列長度)	V3 (最大陣列長度)	ANSI C (最大陣列長度)
一維陣列	256	①	②	不限制
二維陣列	N	①	②	不限制
三或三以上的多維陣列	N	N	②	不限制
指標陣列	N	①	②	不限制
函式陣列	N	功能限制	N	不限制
字符串陣列	不支援	不支援	②	不限制

註：① 若是 const array，則總長度不超過一個 rombank，若是一般的 array 則不超過一個 rambank。

② 若是 const array，則總長度不超過一個 32K，若是一般的 array 則不超過一個 rambank。

4.3 標識符保留字

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	

註：①與②用於定義絕對地址變數，比如：
unsigned char sfr __attribute__ ((at(0x40))); 表示將變數 sfr 定義於 0x40 地址
①與③用於定義中斷向量，比如：
void __attribute((interrupt(0x04))) isr_name(void) {…} 表示在 0x04 地址定義中斷 isr_name
④用于指定入口函式，詳見 2.2.9 節

4.4 運算符

運算符	V1	V2	V3	ANSI C
算術運算符 (+, -, *, /, %)	●	●	●	●
關係運算符 (>, <, ==, >=, <=, !=)	●	●	●	●
邏輯運算符 (!, &&, ||)	●	●	●	●
位運算符 (<<, >>, ~, | , ^ , &)	●	●	●	●
賦值運算符 (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)	●	●	●	●
條件運算符 (? :)	●	●	●	●
逗號運算符 (,)	●	●	●	●
指標運算符 (* 和 &)	●	●	●	●
求字節數運算符 (sizeof)	●	●	●	●
強制類型轉換運算符 (类型))	●	●	●	●
分量運算符 (. ->)	●	●	●	●
下標運算符 ([ ])	●	●	●	●
函式調用運算符 (( ))	●	●	●	●
自增運算符 (++)	●	●	●	●
自減運算符 (--)	●	●	●	●
負號運算符 (-)	●	●	●	●
正號運算符 (+)	●	●	●	●
指定 RAM 變數地址運算符 (@)	●	●		

4.5 前置處理指令

前置處理指令	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

註：① 產出錯誤信息：#error size too big
② HOLTEK C V3 的內嵌彙編使用 asm(“ ”) 格式，詳見 2.4

4.6 預處理指令 #pragma

格式：

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

某些 keyword 會有 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	

註：② HOLTEK C V3 支援 inline 函式，其格式與標準 C 同。

4.7 const 變數

Const 變數功能	V1	V2	V3	ANSI C
適用的數據類型	除 bit	除 bit	除 bit	any
直接被其它文件使用	N	N	Y(引用時，在 const 前加修飾詞 extern)	Y(引用時，在 const 前加修飾詞 extern)
必須宣告為全局型	Y	N	N	N
宣告時要設定初始值	Y	Y	Y	Y
陣列常數要指定陣列的大小	Y	Y	Y	Y
取址操作數	N	N	Y	Y

4.8 預定義的頭檔

預定義的頭檔	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 函式

main 函式的規定	V1	V2	V3	ANSI C
個數 (個)	1	1	1	1
返回數據類型	void	void	void	int
參數 (個)	無	無	無	2 (一個指標陣列)

4.10 中斷函式

中斷函式的規定	V1	V2	V3	ANSI C
設定中斷向量值	Y	Y	Y	沒有中斷函式
個數 (個)	可多個	可多個	可多個	
返回數據類型	void	void	void	
參數	無	無	無	
重覆進入中斷	N	Y ①	Y ①	
在程序中調用中斷	N	N	N	
中斷調用彙編函式	Y	Y	Y	
中斷調用 c 函式	N	Y ②	Y ③	

註：①雖然不同的中斷事件可以重迭發生，但是同一個中斷事件不可以重疊發生，必須等候前一個發生被處理完成後，才能認可下一個中斷事件。針對不具有中斷可重迭 (nested) 發生的微控制器，則在中斷服務函式內不可開啟中斷功能。

②必須將被調用的函式定義成 #pragma nolocal。否則會造成 RAM 空間重疊使用，一般不推薦使用。

③中斷調用的函式不能與 main 函式調用同一個函式，否則會造成 RAM 空間重疊使用，不同的中斷也不能調用同一個函式，比如：

isr1→fun1→fun3

main→fun2→fun1
isr2→fun3
則 isr1 與 main 共同調用了 fun1，isr1 與 isr2 共同調用了 fun3

4.11 內建函式

函式	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
內嵌式彙編語言	Y	Y	Y ②	N
靜態變數	不支援靜態變數和靜態函式	不支援靜態變數和靜態函式	Y	Y
常數	支援二進制常數	支援二進制常數	支援二進制常數	不支援二進制常數
結構體和共享體	bit field 置放於 8 位的單位內，不會橫跨兩個 8 位的單位，且不能定義超過 9 位的 bit field	bit field 置放於 8 位的單位內，不會橫跨兩個 8 位的單位，且不能定義超過 9 位的 bit field	最大可定義 32 位的 bit field	最大可定義 32 位的 bit field
函式	不支援遞歸函式	不支援遞歸函式	不支援遞歸函式	支援遞歸函式
指標	不能用於常數與位變數，不支援函式指標	不能用於常數與位變數，若指向函式，則必須是全域的，且所指函式不能帶有參數	不支援函式指標	Y

功能	V1	V2	V3	ANSI C
初始值	全局變數宣告時不可以同時定義初始值，但是 const 常數在宣告時一定義要設定初始值	全局變數宣告時不可以同時定義初始值，但是 const 常數在宣告時一定義要設定初始值	Y	Y
堆棧	層數有限①	層數有限①	層數有限①	層數不受限制

註：①每個 MCU 的層數有限，調用函式時，要考慮佔用的堆棧層數，一些運算符或函式在調用時所佔用的堆棧層數如下：

運算符 / 函式	堆棧層數	運算符 / 函式	堆棧層數
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	整型與浮點型轉換	1

② HOLTEK C V3 內嵌彙編格式，詳見 2.2.5

第五章 命令列模式

本章主要是使用 C compiler V3 的命令列模式並引導程式員使用命令列模式編譯一個原始檔案。

主要包含如下內容：

- 進入命令列環境
- 使用命令模式生成目標檔的過程
- 命令列參數

5.1 設置環境變數

在使用命令列環境之前，先設置環境變數。

在命令列下使用 set 命令追加 IDE 安裝路徑下的 bin 目錄，例如，

set PATH=%PATH%;XXX/bin

其中 XXX/bin 是 IDE 安裝路徑下的 bin 目錄，這樣就可以在本次的 CMD 視窗中使用 bin 目錄下的各 IDE 工具

5.2 使用命令模式編譯原始檔案的過程

在設置好了環境變數後，就可以開始編寫代碼，這裏以例子 5 為例，把例子 5 中的 file1.c 和 file2.c 編譯成對應的彙編文檔。

命令：hgcc32 [options] cfile –o asmfile

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

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

由上述的指令即可產出編譯後的 asm 文檔。

5.3 命令列參數

參數	含義
-g	產生 debug 信息
-O0/-O1/-O2/-O3/-Os	優化參數
-D<macro>[=<val>]	巨集定義
-I<path>	設定搜尋頭文檔的目錄 path
-msingle-ram-bank	單個 RAM
-mmulti-ram-bank	多個 RAM(default)
-msingle-rom-bank	單個 ROM
-mmulti-rom-bank	多個 ROM(default)
-fno-builtin	不使用 gcc 內建函式
-mno-tbhp	沒有 TBHP
-mtbhp=addr	指定 TBHP 地址 addr(default 為 09H)
-mlong-instruction	擴展指令 MCU

五種優化參數：-O0、-O1、-O2、-O3 和 -Os。編譯時只能設置其中的一種來編譯。

各個優化等級：

-O0：這個等級 (字母“O”後跟個 0) 關閉所有優化選項，也就是沒有設置 -O 等級時的默認等級。這樣就不會優化代碼。

-O1：這是最基本的優化等級。編譯器會在不花費太多時間的同時試圖生成更快

更小的代碼。這些優化是非常基礎的，但一般這些任務肯定能順利完成。

-O2：-O1 的進階優化。這是較優化的等級，-O2 會比 -O1 啟用多一些標記。設置了 -O2 後，編譯器會試圖提高代碼性能而不會增大體積和大量佔用編譯時間。

-O3：這是最高的優化等級。用這個選項會延長編譯代碼的時間，而且有可能會因為這個優化的程度而導致產出的代碼會偏離原來的代碼，一般不使用該優化等級。

-Os：這個等級用來優化代碼尺寸。其中啟用了 -O2 中不會增加存儲空間佔用的代碼生成選項。這對於存儲空間較小的機器非常有用。

更多參數介紹請參考 GCC 使用手冊。

第六章 多文件編程

在一個工程中難免要使用多個原始檔案，把類似的函式和定義編寫到同一個文件中，以方便管理，本章將概要描述多文件編程的相關內容。

主要包含如下內容：

- 頭文件
- 共用的變數
- 調用其他原始檔案中的函式
- 使用庫

6.1 頭文檔

頭文檔用來宣告變數與函式，定義巨集、指定地址變數及類型。不可以定義一般變數及函數。

另外，為了避免頭檔的內容重覆定義，在創建頭檔後需要加上。

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

6.2 共用的變數

如果編程時多個原始檔案需要共同訪部問同一個變數，一般要把該變數定義為全域變數，並且不能加 static，這裏用 extern 的方式宣告該變數是外部檔的變數，這樣就可以使用該變數了；在頭檔中也一樣這麼使用。extern 可以出現在函式體內，也可以出現在函式體之外。

6.3 調用其他原始檔案的函式

使用頭檔的方式和使用 extern 的方式可以引用外部的函式，使用時要註意調用外部的函式時，這些外部的函式可能會修改其所在原始檔案中的全域變數。

例子 12：使用外部函式修改外部變數

代碼清單 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();
    }
}
```

運算結果：sk = 28，g_var = 0x55

6.4 使用函式庫

6.4.1 製作函式庫

請參考《HT-IDE3000 使用手冊》第九章 函式庫總管

6.4.2 制作作函式庫注意事項

不同的 MCU 型號共用函式庫應注意：

1. 指令類型相同，有擴展指令的 MCU 不可與無擴展指令 MCU 共用
2. 單一 ROM/RAM bank 的 MCU 不可與多 ROM/RAM bank 的 MCU 共用
3. 無 TBHP 暫存器與有 TBHP 暫存器的 MCU 不共用
4. ROM 寬度為 16bits 的 MCU 不與 14/15 bits 共用
5. 工程所選擇的參數與製作函式庫時的參數相同

6.4.3 引用函式庫

請參考 2.1.5

第七章 混合語言編程

為了提高程式的效率和 ROM 的利用率，使得合理的利用混合語言編程十分有必要，本章將講述如何使用混合語言來編寫程式。

主要包含如下內容：

- 數據存儲格式
- C 語言調用彙編語言函式
- 彙編語言調用 C 語言函式

7.1 數據存儲格式

採用小端數存儲模式 (Little Endian)，即一個數據的低字節存放在低地址處，而高字節存放在高地址處，例如：

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

數據在存儲器中的存放結果如下：

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

7.2 變數，函式的命名規則

1、Holtek C V3 編譯器在編譯全局變數 (global variable) 及函式時，會在原名之前附加一個底線字符 (underscore)，例如：

全局變數 count 編譯後改為 _count

函式 GetTotalSize 編譯後改為 _GetTotalSize

局部變數，靜態變數，函式參數等命名則比較不規則，可以參考其編譯出的彙編文件中的 debug 信息，但要註意，每次編譯後的名字有可能不同。

2、彙編器 (Assembler) 在編譯彙編程序後，會將名字轉換成大寫字母 (upper case)，例如：

變數 count 編譯後改為 COUNT

函式 GetTotalSize 編譯後改為 GETTOTALSIZE

7.3 C 語言調用彙編語言函式

彙編語言程式的函式定義規則：

1、將底線字母 (underscore) 加入函式名字之前，並且宣告為公用變數 (public variable)。

2、如果函式具有參數，將其宣告為公用變數。

3、函式中若有局部變數，使用 local 定義。

4、使用 proc/endp 定義函式。

C 程式的調用規則：

1、以大寫字母定義並宣告要調用的函式名字。

2、若被調用的函式具有參數， 則要外部宣告參數。

3、調用此函式。

例子 13：彙編語言減法函式

代碼清單 6.2：

```
CODE.ASM
public _opera
public _opera_var1
public _opera_var2                ;; 將函式及參數定義成 public，方便外部調用

_opera .section page 'code'
_opera proc                       ;; 使用 proc/endp 定義函式
   local _opera_var1      db ?    ;; 參數定義
   local _opera_var2      db ?
   local _result_local    db ?    ;; 局部變數定義
   mov a, _opera_var1
   sub a, _opera_var2
   mov _result_local, a
   ret
_opera endp

MAIN.C
extern unsigned char OPERA();     // 以大寫字母定義函式名字
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();              // 函式調用
   while(1){
       asm("nop");
   }
}
```

執行結果：result = 0x10

7.4 彙編語言調用 C 語言函式

C 程式的調用規則：以大寫字母定義並宣告被調用的函式名字。

組合語言程式的函式定義規則：

1、將函式名字宣告為外部名字，需在函式名前加底線“_”。

2、調用函式用 proc/endp 聲明。

3、如果函式具有參數，則將所對應的變數宣告為外部變數，參數的彙編名字，可以參考 C 函式所編譯出的彙編檔，註意，每次編譯的結果可能不同，所以盡量不要帶參數。

4、調用 C 函式後，讀出返回值，若返回值為 1byte，則存於 ACC，若為 2byte，則低字節存於 ra，高字節存於 rb，若為四 byte，由低字節到高字節分別存於 ra、rb、rc、rd，其中 ra~rd 都已定義好，只需宣告就可以使用。

代碼清單 6.3：

```
FUN.C
int DISPLAY(char row, int col)    // 定義函式，函式名必須大寫
{
  int retval;
  retval=(int)(row << 1) + col;
  return retval;
}
CODE.ASM
                                  ;;code.asm 調用函式 _DISPLAY
extern _DISPLAY : near            ;; 函式宣告為外部名字
extern _DISPLAY_2 : byte          ;; 宣告參數變數名
extern ra : byte                  ;; 宣告返回值
extern rb : byte
CODE .section 'code'
_code proc
local _code_loc db 2 dup(?)       ;; 局部變數定義
MOV A, 10h
MOV _DISPLAY_2, A                 ;; 存值到第二個參數 col 的低位元組
CLR _DISPLAY_2[1]                 ;; 第二個參數的高位元組設為 0
MOV A, 20h
MOV _DISPLAY_2[2], A              ;; 存值到第一個參數 row
CALL _DISPLAY                     ;; 調用 C 函式 _DISPLAY
MOV A,ra
MOV _code_loc,A
MOV A,rb
MOV _code_loc[1],A                ;; 將返回值從 ra,rb… 讀出，存入局部變
                                  ;; 量 _code_loc，低字節 ra，高字節 rb
RET
_code endp
```

運算結果：變數 code_loc = 0x0050

註意如果鏈接時出現如圖 6_3_1 的錯誤所示：

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

圖 6_3_1

則需要在編譯參數中勾選“Case sensitive for assembly”選項。

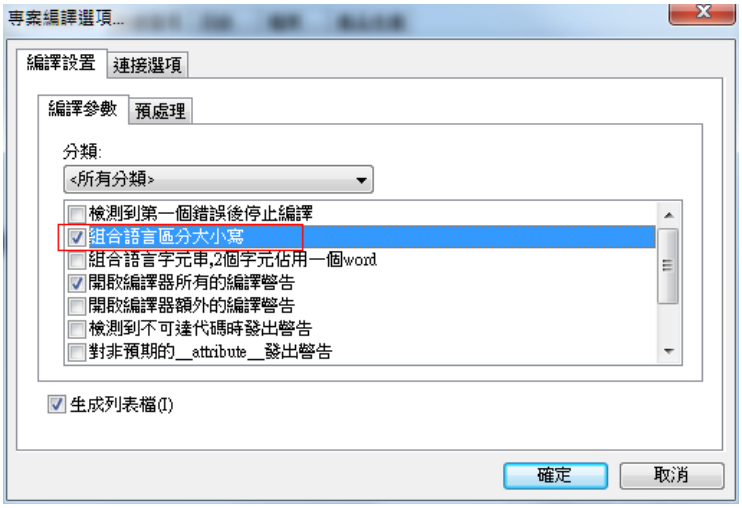


圖 6_3_2

第八章 常見錯誤的解決方式

8.1 內部錯誤 (Internal Error)

若 error 信息有 internal compiler error 字樣，則為 compiler 內部錯誤，請與 Holtek 公司反饋，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

對于無擴展指令架構的 MCU，C Compiler 會默認把變數配置到 RAM bank0 (有擴展指令的 MCU 可以自動配置到任意 bank)，當 bank0 滿了之後，會報 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)
```

出現此信息後，做法如下：

- 檢查資料類型是否誤用 (特別是從 V1 C Compiler 移植過來的程式)
- 若為 multi RAM bank MCU, 可手動將全域變數調到其它 bank，參考 2.2.2 節

8.3 ROM/RAM 空間溢出

當 ROM 或 RAM 空間不夠時，會出現這個信息：

```
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)
```

出現此信息後，做法如下：

- 檢查是否打開優化參數 -Os，參考 2.1.4 節。
- 查看 map 檔，了解 RAM/ROM 分配情況，刪減不必要的程式。

8.4 變數重疊警告

絕對位址變數位置重疊，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)
```

出現此 warning 有兩種可能情況：

- 同一個絕對地址變數在不同檔定義多次，比如，在 a.h 中定義變數 var：

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

  在 t1.c 與 t2.c 同時 include a.h 就會報這個 warning，對于這種情況，此 waring 信息可以忽略，也可以通過選項設定避免此 warning，參考 2.1.5 節
- 不同變數定義的位址重疊，如下，_b 與 _a 位址重疊，需將 _b 定義在 0x0142

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

8.5 變數重定義

如果一個變數 (非絕對位址變數) 定義在頭文檔，而這個頭文檔被多個 .c 文檔引用，則會出現變數重定義，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
```

解決方式：

不要在標頭檔中定義變數，若 t.c 與 t1.c 都需要用到 a，則在其中一個檔定義 a，在標頭中聲明 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 程式範例

本章將使用 C compiler V3 編寫常見的 MCU 程式，以便快速使用 C compiler V3 進行工程開發。

涵蓋如下主要內容：

- 中斷函式的使用
- 混合編程的用法

9.1 使用中斷讓 LED 燈閃爍

使用定時器讓 LED 燈閃爍，時間間隔為 1s，即亮 1s 鐘後熄滅 1s 鐘。

代碼清單 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 使用表格讓 7 節 LED 管顯示數字

這裏是用混合編程的形式從彙編表格中讀出數據顯示在 7 節 LED 管上。

代碼清單 7.2：

```
Table.ASM
#INCLUDE HT66F50.INC
public _code

_code .SECTION 'CODE'
_code proc
```

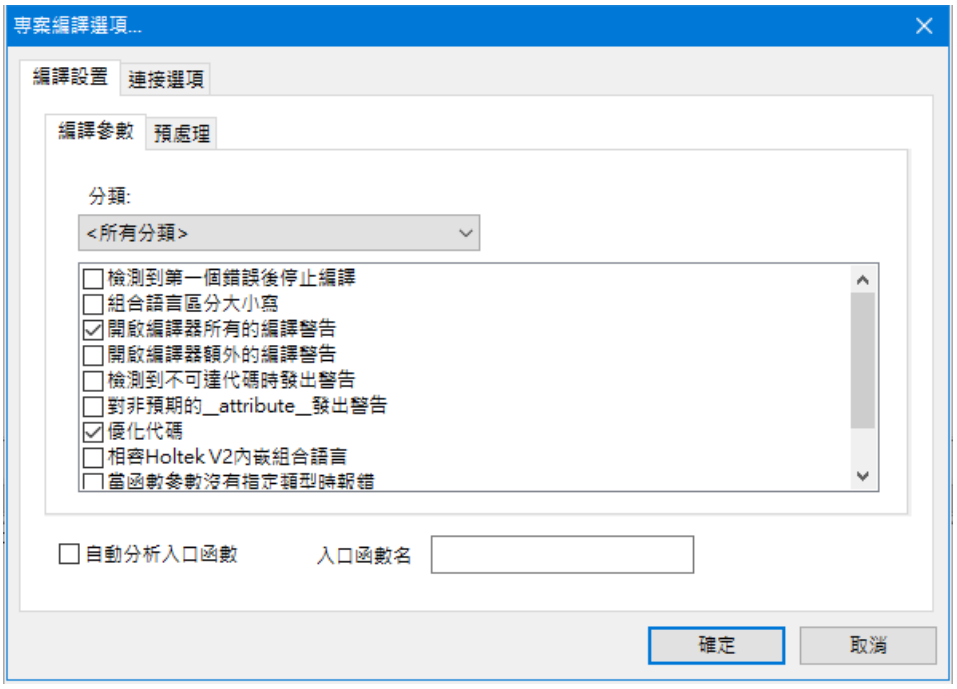
```
TAB_7_SEG:
   DC 0F9C0H
   DC 0B0A4H
   DC 09299H
   DC 0F882H
   DC 09580H
   DC 08388H
   DC 0A1A7H
   DC 08E86H
_code endp
   END
Main.c
#include "HT66F50.h"
void _delay(unsigned char times)
{
   volatile unsigned char t1, t2, t3;
   t1 = times;
   while(t1--){
         t2 = 3;
         while(t2--){
               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 程式優化寫法

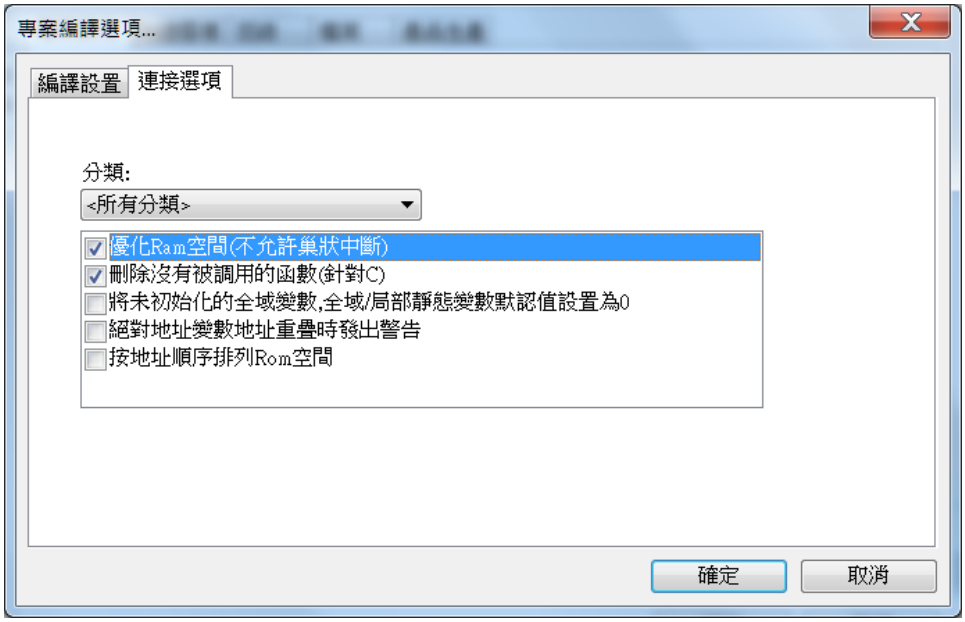
10.1 優化選項

選擇“優化代碼”，即 -Os，如下圖，compiler 會生成最優化的代碼，具體的優化內容參考第三章內容。

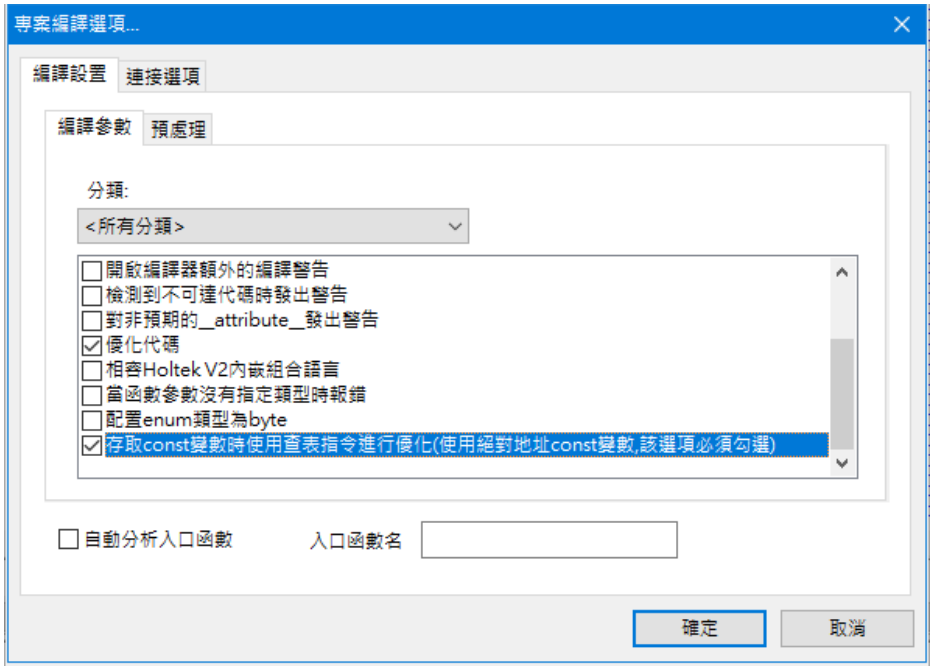


如果整個程式中的中斷服務程式沒有發生嵌套，即執行一個中斷時不會進入另一個中斷，則可以選擇以下“優化 RAM 空間”參數，節省 RAM 空間。

中斷的程式要儘量簡短，否則會佔用太多的 RAM。

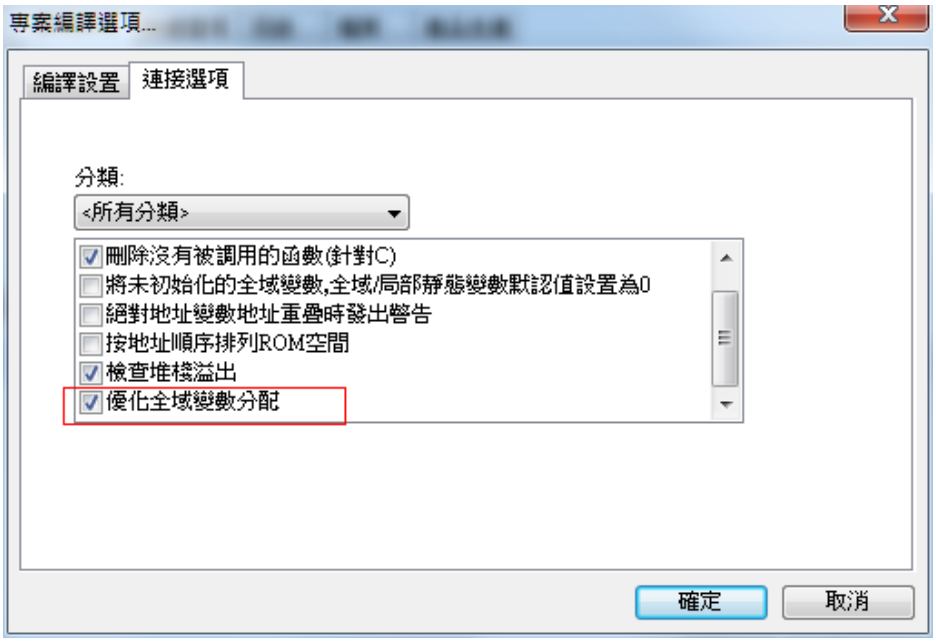


對具有 TBHP 且 ROM 寬度為 16bit 的無擴展指令 MCU，編譯器支援用查表指令的方式存取 const 變數，如下參數 (此參數默認勾選)，如果 const 變數個數很多，勾選此參數會節省 code，反之，則會浪費 code，用戶可以根據自己的實際程式選擇是否勾選。

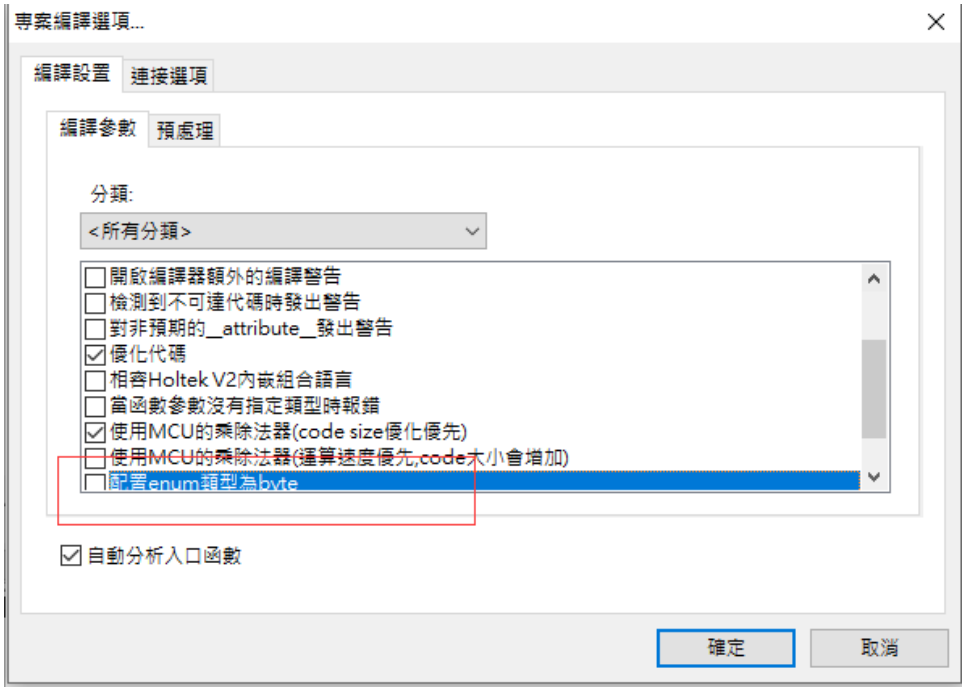


可以選擇以下參數“優化全域變數分配”：

具擴展指令架構的 MCU，不需要指定變數地址，連接器會自動分配所有的變數地址 (包括 RAM bank0 以外)，按變數的使用頻率分配，優先分配次數多的變數到 bank0。



若定義的 enum 資料不超過 127，或勾選以下選項：



10.2 變數聲明

10.2.1 unsigned/signed

在不會出現負數的環境中，使用 unsigned 會更節省 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 資料形態

選擇適當範圍的資料類型，助於產生更精簡的指令。

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 浮點常數

浮點常數默認為 double 類型，如果計算所需要的精度不需太高，可以將它強制轉為 float，如：(float)3.14 .

float s,r;
void main()
{
  s = r * r * 3.14;
}
	float s,r;
	void main()
	{
	  s = r * r * (float)3.14;
	}

Code size:343	Code size:177

編譯器不會給浮點運算常量折叠，所以，如果有兩個浮點常數需要計算，可以把結果先算出來：

#define HALF (float)0.5
#define QUARTER (float)0.25
float l,r;
void main()
{
  r = l * HALF * HALF;
}
	#define HALF (float)0.5
	#define QUARTER (float)0.25
	float l,r;
	void main()
	{
	  r = l * QUARTER;
	}

Code size:171	Code size:152

10.2.4 const 陣列

把 const 陣列定義成全域會比局域的節省 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];
}
	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:23	RAM size:16

10.2.5 將功能相似的變數定義成陣列以便使用迴圈語句

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);
  }
}
	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:140	Code size:75

10.2.6 除 delay 函數外，區域變數不使用 volatile 修飾

10.3 程式結構

10.3.1 調整語句順序以適用尾部合併優化

編譯器會做尾部合併優化，參考 3.12 節，對於 switch 或 if/else 語句，可以將相同的語句寫在分支的後面，以適用尾部合併。

```
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 重複多次的運算可以用迴圈代替

當程式中存在多次重複的運算，而且都有規律性，可以用迴圈代替。

```
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 函式呼叫

10.4.1 避免不必要的函式呼叫

若某函式被多次調用，但其返回值並無差異，應考慮用變數接收函式返回值，使用時做為替代。

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;
	}

	減少運行時間

10.4.2 封裝頻繁使用的代碼為函數

如果在程式中存在某段代碼被多次使用，可將這段代碼封裝成為函數，以便節省指令。

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 如果函式只在本文檔調用，可以定義成 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 全局變數的分配

對無擴展指令架構的 MCU，RAM BANK0 以外的地址只能用間接定址的方式訪問，下例可以看出，間接定址的指令比直接定址多出 5 條，所以，當 RAM bank0 溢出時，用戶可以選擇把較少用到的變數定義到其它 bank，比較常用的變數定義在 bank0。

直接定址 (Bank 0)	間接定址 (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 中斷服務程式

一般，如果兩個函式沒有調用關係，那它們的局域變數是可以分配到同樣的地址，但中斷服務程式不會與主函式共用局域變數的地址，所以，為了減少 RAM 的使用，中斷程式可以儘量簡單，不宜寫得太過複雜。

10.7 變數初始化

若程式中已經有寫 CLR RAM 的函數，可將全域 / static 變數定義的初始值去掉，因為此時初始值無效。

附錄 A：ASCII 碼表

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	

附錄 B：運算優先級

優先級	運算符	含義	運算類型	結合性
1	( )	圓括號	單目	自左向右
	[ ]	下標運算符		
	->	指向結構體成員運算符		
	.	結構體成員運算符		
2	!	邏輯非運算符	單目	自右向左
	~	按位取反運算符		
	++ --	自增、自減運算符		
	(類型關鍵字)	強制類型轉換		
	+ -	正、負號運算符		
	*	指標運算符		
	&	地址運算符		
	sizeof	長度運算符		
3	* / %	乘、除、求余運算符	雙目	自左向右
4	+ -	加、減運算符	雙目	自左向右
5	<<	左移運算符	雙目	自左向右
	>>	右移運算符		
6	< <= > >=	小於、小於等於、大於、大於等於	關係	自左向右
7	== ! =	等於、不等於	關係	自左向右
8	&	按位與運算符	位運算	自左向右
9	^	按位異或運算符	位運算	自左向右
10	|	按位或運算符	位運算	自左向右
11	&&	邏輯與運算符	位運算	自左向右
12	||	邏輯或運算符	位運算	自左向右
13	? :	條件運算符	三目	自右向左
14	= += -= *= /= %= << = >>= &= ^= |=	賦值運算符	雙目	自右向左
15	,	逗號運算符	順序	自左向右

附錄 C：命令列模式命令參數及功能

1. compiler 參數

命令：hgcc32 [options] cfile –o asmfile

參數	含義
-g	產生 debug 信息
-O0/-O1/-O2/-O3/-Os	優化參數
-D<macro>[=<val>]	巨集定義
-I<path>	設定搜尋頭文檔的目錄 path
-msingle-ram-bank	單個 RAM
-mmulti-ram-bank	多個 RAM(default)
-msingle-rom-bank	單個 ROM
-mmulti-rom-bank	多個 ROM(default)
-fno-builtin	不使用 gcc 內建函式
-mno-tbhp	沒有 TBHP
-mtbhp=addr	指定 TBHP 地址 addr(default 為 1fH)
-mlong-instruction	擴展指令 MCU

2. assembler 參數

命令：hasmgcc32 [options] source, object, listing

參數	含義
/chip=chip-name	指定 MCU 型號
/case	區分大小寫
/d<macro>	巨集定義
/i<include-path>	設定搜尋頭文檔的目錄 path
/z	產生 debug 信息
/h (/?)	顯示幫助
source	要編譯的 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 參數

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]]]] [;]

註意：以上的參數一定要按照上面的順序設置，否則 linker 會出錯或不能正確執行，而且參數是區分大小寫的。

環境變數 LIB 為搜尋 lib 文件的目錄。

參數	含義
/HIDE	IDE 的控制碼。8 個 16 進制數
/MCU	工程使用的 MCU 名稱
/NOLOGO	隱藏 LOGO
/novectornest	有該參數表示 ISR 不會嵌套， 可以節省 RAM space (for C Compiler V3 only)
/OptimizeParam=x	默認為 0， x 的低半位元表示 BP 優化參數，0 表示關閉， 2 表示開啟 x 的高半位元表示 Dead Section 優化，0 表示關閉， 1 表示開啟 (for C Compiler V3 only)
/OptimizeLInst=x	表示擴展指令優化，0 表示不優化，1 表示優化。 默認不優化 (for C Compiler V3 only)
/Startup0	將沒有初始值的全域變數初始化為 0 (for C Compiler V3 only)
/TBHP=x	x 表示 TBHP 寄存器的位址，默認位址為 9
/EEPROM=xx	指定 EEPROM_DATA_SIZE，默認為 0 (for C Compiler V3 only)
/ERRORLOG	保存錯誤日誌的路徑
/MAP	用戶指定必須生成 MAP 檔
/ADDR:section_name=addr [,section_name=addr]	指定某些 section 的分配地址由指定地址 address 開始。 註：addr 為 16 進制
/HELP (/?)	顯示命令列格式幫助資訊

註意：調用 assembler 和 linker 前先設置環境變數 CFG。

set HTCFG=IDE 目錄 \ MCU。

參考讀物

《HT-IDE3000 使用手冊》

介紹 HT-IDE，assembler，linker 等 tool 的使用，可于 HT-IDE3000 安裝檔 DOC 目錄下取得。

《Holtek 標準函式庫使用手冊》

介紹 Holtek C 支援的標準函式庫及使用方式，可于 HT-IDE3000 安裝檔 DOC 目錄下取得。

《Holtek C Compiler V3 FAQ》

C Compiler V3 常見的 FAQ 問題，持續更新中，可于 HT-IDE3000 安裝檔 DOC 目錄下取得。

《gcc manual》

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