TM89 C 編譯器

4-位元 微型控制器

使用手冊

AMENDMENT HISTORY

Version	Date	Description
V1.0	Oct, 2011	新頒

內容

1. TM89 系列 C 語言編譯器概述

關於 TM89 系列 C 語言編譯器

TM89 系列 C 語言編譯器符合 ANSI C 標準，可是 TM89 系列 C 語言編譯器不支援函式指標。

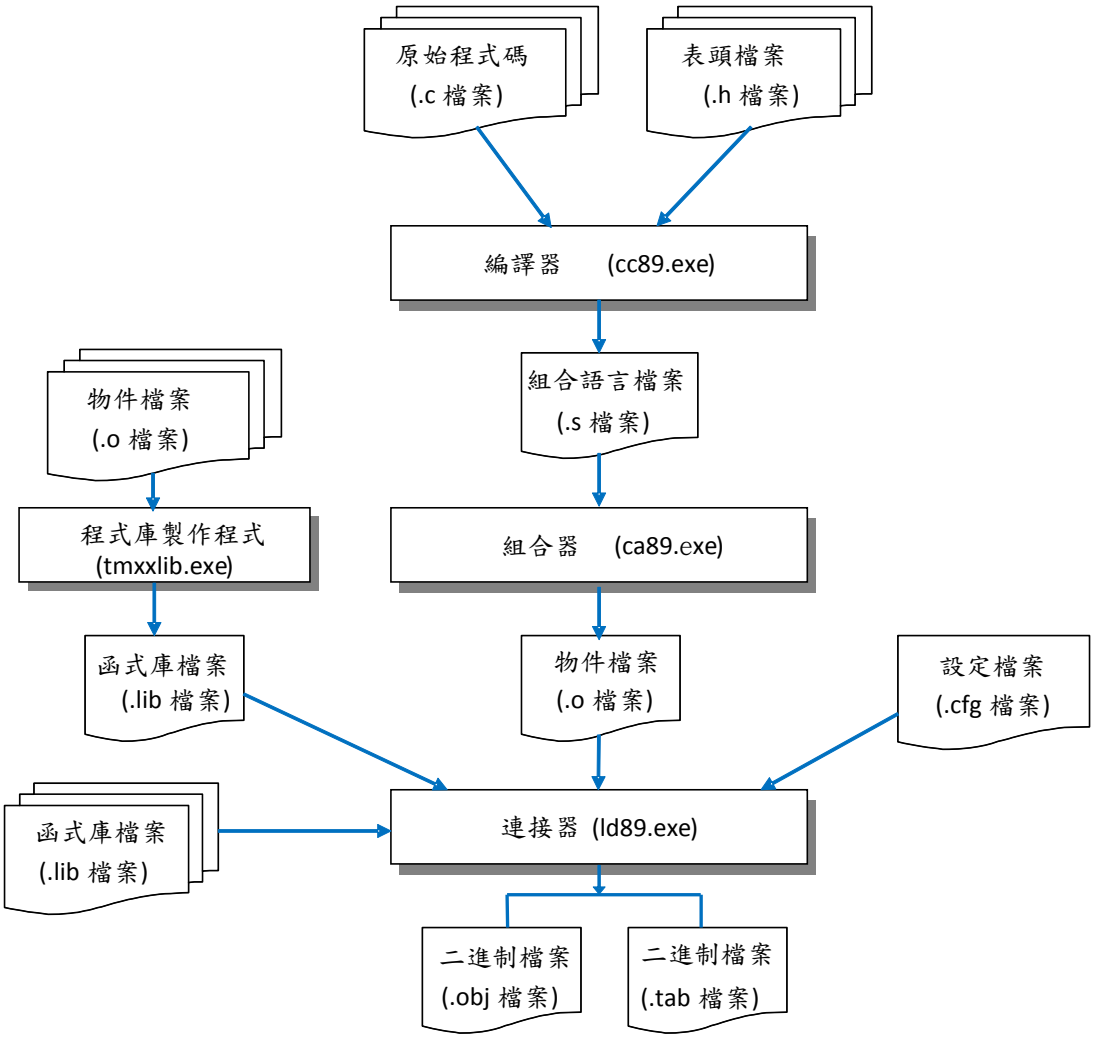
編譯 C 語言程式

微型控制器程式一定要適合現有的晶片內程式記憶體。提供系統外部或可擴充記憶體將提高成本。編譯器和組合器是用來轉換高階語言和組合語言變成一個較小型的機器語言存放到微型控制器的記憶體。

在編譯 C 語言程式時，您會接觸以下五種檔案類型：

1. 一般的原始程式碼檔案。此檔案包含函式定義，照慣例，此檔案會以".c" 為副檔名。
2. 表頭檔案。這些檔案包含函式宣告（亦稱為函式原型）和各類前處理器句子。這些檔案可以讓原始程式碼使用外部定義的函式。表頭檔是以".h"為副檔名。
3. 物件檔案。這些檔案是編譯器的輸出檔案。他們包含以二進制的函式定義，可是這些是無法獨立執行的檔案格式。物件檔案的副檔名為".o"。
4. 晶片相關檔案，包含執行中使用的程式庫檔（runtime_89.lib 檔案）和設定檔案（.cfg 檔案）。這些檔案包含記憶體位址重新配置和晶片指令集之資訊。
5. 二進制可執行的檔案。這些檔案是由一個叫"linker"程式產生的。Linker（連接器）會把多個目標檔案作連接並產生一個二進制可直接執行的檔案。二進制可執行檔的副檔名為".bin"。

還有其他類型的檔案，如組合語言檔案（".s"檔案）和變數資訊檔案（".cfn"檔案），可是一般情況您不需要直接處理這些檔案。


原始程式碼
(.c 檔案)
表頭檔案
(.h 檔案)
編譯器 (cc89.exe)
物件檔案
(.o 檔案)
組合語言檔案
(.s 檔案)
程式庫製作程式
(tmxxlib.exe)
組合器 (ca89.exe)
函式庫檔案
(.lib 檔案)
物件檔案
(.o 檔案)
設定檔案
(.cfg 檔案)
函式庫檔案
(.lib 檔案)
連接器 (ld89.exe)
二進制檔案
(.obj 檔案)
二進制檔案
(.tab 檔案)

詞彙約定

一個詞彙元素指一個字元或字元群體可以合法出現在原始程式碼檔案中。此章節將討論 C 語言詞彙約定包含 tokens、字元集（character sets）、註譯（comments）、識別符號（identifiers）和常數字符（constant literals）。

Tokens 是一系列連續的字元，C 語言編譯器把它當作一個數據單元。空格、移字符號、新行、註解，這些整體都被當作是"空白字元"。空白字元將被 C 語言編譯器忽略除非它是用在區分 tokens。有些空白字元是必須的，尤其是它可以隔開相鄰的識別符號、關鍵字和常數。適當的空白字元將使程式碼更容易閱讀與維護。

有六種不同類別的 token：

- 識別符號（Identifiers）
- 關鍵字（Keywords）
- 常數（Constant）
- 字母符號（Literals）
- 運算符號（Operators）
- 標點符號（Punctuators）

原始程式字元集

以下列出可用於編譯和執行時期的基本原始字元集：

- 大寫和小寫英文字母

  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- 十進位數字 0~9

  0 1 2 3 4 5 6 7 8 9

- 底線字元（_）
- 以下標點符號（標點符號是具有語法和語意含義的字元）

!	#	&	(	{
“	%	’	)	}
,	-	.	/	
;	=	[	]	
<	>	\	_	
~	*	+	:	

- 空白字元
- 脫離序列（Escape Sequences）：一些特殊並且非圖型符號會以脫離序列來代表。

脫離序列的意義代表如下：

脫離序列	代表的字元
\b	退格鍵（Backspace）
\f	換頁
\n	換行
\r	回車
\t	水平制表
\v	垂直制表
\’	單引號
\”	雙引號
\\	反斜線

註譯

在前置處理過程中，註譯會以空白字元代替；編譯器因此忽略所有註譯。

有兩種註譯：

- */** （斜線，星號）字元開始一段註譯，後面會隨著任何字元（包含新行），並且以*/結束此段註譯。這種註譯通常被稱為 *C-型式的註譯*。
- *//* （雙斜線）字元後面隨著任何字元。前面沒有直接以反斜線起始的新行將終止這類的註譯。這種註譯通常被稱為 *單行註譯*。

注意: 您不能在 C-型註譯裡面再使用 C-型註譯。意思是指每一個註譯將以第一次出現的 */ 當結尾。可是您可以使用單行註譯在 C-型註譯當中。

識別符號

識別符號或名稱，包含任意數量的字母、數字或底線（_）。第一個字元不可以為數字。大寫和小寫字母是有區別的。C 語言編譯器會區分識別符號的大寫和小寫。例如，*TENX* 和 *tenx* 代表不同的識別符號。

識別符號提供名稱給以下語言元素：

- 函式
- 物件
- 標籤
- 函式參數
- 巨集和巨集參數
- 型態宣告
- 結構（struct）和聯合（union）名稱

關鍵字

關鍵字是一個為了特殊用途所預留的識別符號。您也可以做為前置處理的巨集名稱，但是此為不好的撰寫程式的型式。只有精確拼字的關鍵字是被預留的。為了擴展 C 語言能力以達到 MCU 的特質，已增加一些額外的關鍵字到 TM89 系列 C 語言編譯器。以下列出此編譯器所預留的關鍵字：

asm	enum	short	void
break	extern	signed	while
case	float	static	**__asm__**
char	for	struct	**interrupt**
continue	goto	switch	**nibble**
default	if	typedef	unsigned
do	int	union	return
else	long		

注意：

- ***__asm__ 、interrupt 和 nibble 為額外 MCU 的關鍵字***
  - __asm__
  - interrupt
  - nibble
- ***不支援 double***

常數

常數是不可定址的，意思是指此數值儲存在記憶體的某處，可是我們沒有必要去存取那記憶體位址。每一個常數包含其數值和資料類別。

任何常數的值在程式執行中是不會改變的，並且一定要在其類別可代表數值的範圍內。以下為常數可用的類別：

- 數字常數
- 字元常數
- 列舉常數

數字常數

數字常數可以用十進制、十六進制和八進制來表示，可根據字首來識別。

八進制常數是啟始數字為 0 的一系列數字。八進制常數僅包含數字 0 到 7。一系列數字前面以 0x 或 0X 啟始的是十六進制的整數。十六進制數字包含 [aA] 到 [fF]，其值為 10 到 15。字尾[L]或[l]習慣上表示數字常數的資料型態為長整數（long）。此字尾雖然是被允許的，可是是多餘的，但此寫法將使程式碼更容易被理解。

- 語法：
  十進制：預設
  十六進制常數：數字以 “0x” 為字首
  八進制常數：數字以 “0” 為字首

- 範例：

```
12, 34              // 十進制常數
0x5A, 0xB2          // 十六進制常數
014                 // 八進制常數
3452L              // 長整數常數
```

注意：不支援二進制常數

字元常數

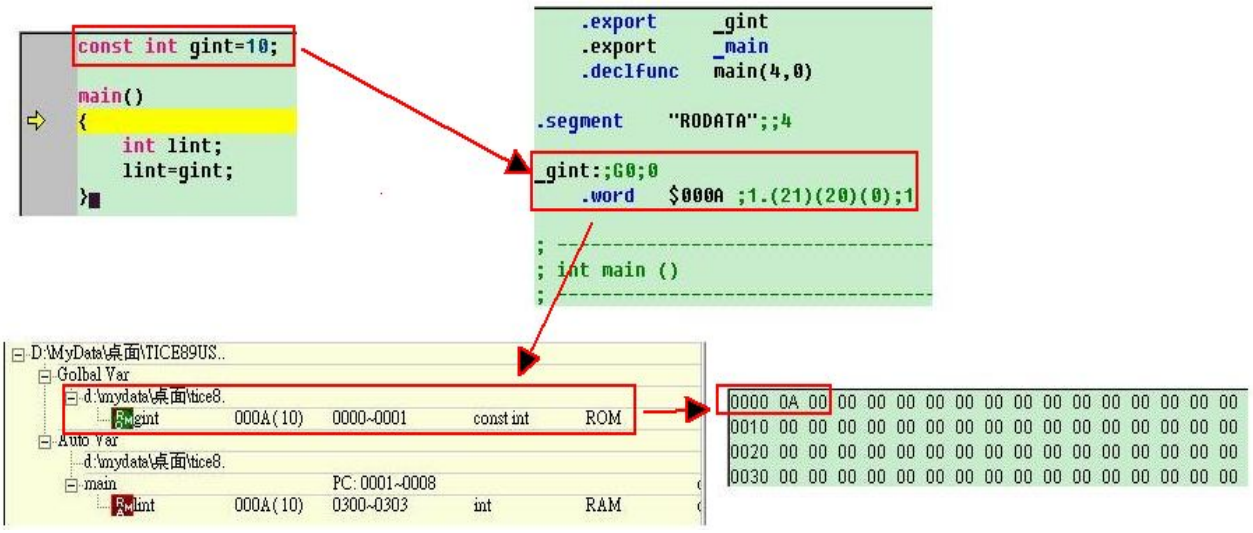
一個字元常數是一個字元以單引號圍住的，例如 'x'。字元常數的值為機器字元集的數字值。字元常數的型態為整數。

列舉常數

在 ANSI C 裡，列舉常數為可在任何地方使用的常整數。意思是指被宣告為列舉的名稱擁有整數的型態。同樣的，ANSI C 允許其他整數型態的變數被賦值為列舉型態的變數。

全域常數

在 TM89 系列 C 語言編譯器，全域常數變數儲存在 TABLE ROM（程式記憶體）。例如，宣告一個全域常數：`const int gInt=10;` 編譯器分配變數的地址在 0000~0001。以此為例，變數數據為 0x000A = 10。



字串常數

字串常數為以雙引號圍著的一系列字元，如“...”。字串常數的型態為字元矩陣且其初始值是被定義的。TM89 系列 C 語言編譯器將擺一個空位元（\0）在字串常數的結尾讓程式在掃字串常數時找到其終點。另外，也可以使用前面提到的脫離序列字元常數（請參考 ”原始程式字元集” 查看脫離序列的列表）。

運算符號

運算符號說明將進行的運算。運算符號 [ ] 和 ( ) 必須為配對出現，也可以被其他式子隔開來。運算符號為下列：

[ ]( ). ->
++ - - & * + - ~ ! sizeof
/ % << >> < > <= >= == != ^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=

標點符號

標點符號是一個含有語意的符號，可是沒有說明將要進行的運算。標點符號 [ ]、( ) 和 { }必須以一對來呈現，但也可能被運算式、宣告或句子隔開。標點符號如下列：

[ ]( ){ } * , : = ; ... #

有一些運算符號，根據內容也算是標點符號。例如，矩陣所引指標 [ ] 為一個宣告的標點符號。

2. 識別符號的意義

ANSI C 識別符號由以下四種特性來消除歧異：*作用域（scope）*、*命名空間（name space）*、*鏈結（linkage）*和*儲存空間的持續時間 (storage duration)*。在此章節中將只討論，存儲類別說明符對一個物件的儲存空間持續時間和鏈結之影響。

消除歧異的名稱

此章節討論 C 語言裡消取名稱歧異的方式：*作用域*、*命名空間*、*鏈結和儲存空間的持續時間*。

作用域（Scope）

程式裡最大區域，在此可看見識別符號並且可以有效的被用來參考其物件，亦稱為識別符號的作用域（***scope of identifier***）。編譯器依據作用域的規則和名稱分析來判斷檔案裡面的識別符號是否在此判斷點為合法。

區塊作用域（Block Scope）

區塊作用域是自動變數的作用域，宣告於一函式或一區塊之中。在不同的區塊中宣告相同的識別符號將不會造成衝突。當某一區塊包含在另外一個區塊之中，外面區塊的識別符號可以被內層包圍的區塊中看到。而在內層區塊中的識別符號將被隱藏，直到程式執行完內層區塊內的所有程式。當控制程式回到外層區塊時，將恢復外部識別符號的宣告。此情況稱為區塊可見度（***block visibility***）。

函式作用域（Function Scope）

只有標籤（***labels***）有函式作用域。標籤是被宣告於其程式本文之中，並且可持續被看見直到宣告標籤的函數結束。標籤可用在 goto 陳述句之中，以跳轉到宣告標籤的程式段。

函式原型作用域

若一個識別符號出現在函式原型（function prototype）的參數宣告中，但其不屬於函式定義的一部份時，他就有函式原型作用域。函式原型作用域的作用範圍結束於原型宣告之後。

可參考以下例子：

```
char * getEnvName (const char * name);
int name;
```

int 型態變數: name 與函數參數: name 將不會造成衝突，因為函數參數 name 的作用域結束於函數原型宣告以外。然而，函數原型仍在作用域之中。

檔案作用域（全域作用域）

檔案作用域（或全域作用域）應用在任何區塊、函式或函式原型宣告之外的識別符號。具有全域作用域和內部鏈結的識別符號，可見的範圍由其符號宣告的地方直到轉譯單元的結束處。

具有全域作用域的識別符號也可以被存取作為全域變數的初始化。若識別符號被宣告為 ***extern***，則此符號在連結所有目標檔案過程中是可被看見的。

命名空間中的識別符號

C 語言編譯器建立命名空間來區分不同的識別符號，並對應到不同的實體之中。同樣的識別符號在不同的命名空間不會互相干擾，即使他們都在同樣的作用域。您可以在同樣的命名空間中，在巢狀的程式區塊之中重覆定義相同的識別符號。

ANSI C 認定以下四種不同的命名空間：

- **Tags**：*struct*、 *union* 和 *enum* tags 為同一個命名空間
- **Labels**：*labels* 是位於他們自己的命名空間
- **Members**：每個 *struct* 或 *union* 的成員有個自的命名空間
- **普通的識別符號**：以下識別符號在單一個作用域裡面必須是唯一的
  - C 函數名稱
  - 變數名稱
  - 函數參數的名稱
  - 列舉常數
  - 型態宣告的名稱

識別符號的鏈結

鏈結是指跨過多個或在同一個編譯單元內，識別符號的可用性或有效性。翻譯單元指一個原始程式碼檔案加上所有表頭和其他原始檔案，包含經前置處理過後的 #include 指令，再扣掉任何被省略的程式碼（因為有一些條件式的前置處理指令）。鏈結允許識別符號的實例（instance）正確的與一個特定的物件或函式做結合。

儲存空間持續期間（Storage Duration）

識別符號的作用域與物件的儲存持續期間是相互關係的，此時間指的是一個物件可以保留在某特定的儲存區域多久的時間。物件的壽命受儲存空間持續期間所影響，同時也被物件識別符號的作用域所影響。

3. 宣告（Declaration）

宣告 決定物件當中相互關連的屬性：儲存類別、型態、作用域、可見度、儲存空間持續期間和鏈結。

宣告的表示方式如下：

declaration:	*declaration-specifiers [init-declarator-list]*

declaration-specifiers 包含一系列的說明其決定宣告中識別符號的鏈結、儲存空間持續期間和型態。

Declaration-specifiers:	*storage-class-specifier [declaration-specifiers]* *type-specifier [declaration-specifiers]* *type-qualifier [declaration-specifiers]*

init-declarator-list 是一系列以逗點符號隔開的宣告而且這是非必需的，每一個都可以有初始程序。

Init-declarator-list:	*init-declarator* *init-declarator-list , init-declarator*
init-declarator:	*Declarator* *declarator = initializer*

宣告決定以下物件資料的屬性和他們的識別符號：

- **作用域 (Scope)**，識別符號可以存取其物件之程式碼區域。
- **可見度 (Visibility)**，可以合法存取識別符號的物件之程式碼的區域。
- **持續期間 (Duration)**，識別符號的真實或實際物件，其存在記憶體中所持續的期間。
- **鏈結 (Linkage)**，識別符號和特定物件之間有正確的鏈接。
- **型態 (Type)**，表示實際有多少記憶體可配置給物件。

儲存類別說明符

儲存類別說明符指示了鏈結和儲存持續期間。儲存類別說明符 的表示方式如下：

storage-class-specifier:	static extern typedef

typedef 沒有預留儲存空間，而且為了語法方便被歸類為儲存類別說明符（storage-class specifier）。

以下為應用在儲存類別說明符的使用規則：

- 一個宣告最多可以有一個儲存類別說明符。若儲存類別說明符不存在，此儲存空間只有在定義該物件的程式區塊之執行期間才可被維護（意即為自動變數）。
- 一個函式中的識別符以儲存類別 **extern** 宣告時，其必有一外部連結，意思是指其他編譯單位可以呼叫此識別符。
- 識別符以儲存類別 **static** 宣告時，其有靜態的儲存持續期間，和內部鏈結（若在函式外部宣告）或沒有鏈結（在函式內部宣告）。若要對識別符初始化，初始化語句將會在執行前做一次。若沒有明確的初始化，靜態物件將被內隱初始化為 0。

型態說明符（Type Specifiers）

型態說明符之語法如下：

type-specifier:	*struct-or-union-specifier*
	typedef-name
	enum-specifier
	char
	short
	int
	long
	float
	nibble
	signed
	unsigned
	void

注意：不支援資料型態 *double*

半位元組（nibble）

資料型態：半位元組是一 4-位元的聚合或是半個位元組，在 4-位元微控制器，半位元組是用以形容儲存數位數值資料的記憶體數量， 即半位元組是一基本資料單元。利用此項技術以加速計算速度。在 C 語言中，半位元組可以是有號數（signed） 或無號數（unsigned）。

注意：雖然半位元組是 4-位元的資料型態，但 TM89 C 編譯器為增加資料讀/寫的速度，仍賦於它 8-位元的空間以儲存資料。

以下例子為使用半位元組指標變數來控制 LCD RAM 的存取：

```
nibble *lcd_ram;          //宣告一個 nibble pointer
lcd_ram = 0x0100;         //指定 lcd_ram 的位址為 0x0100
*lcd_ram = 0xF;
lcd_ram = lcd_ram+1;      //指定 lcd_ram 的位址為 0x0102
lcd_ram = 0x0100+1;       //指定 lcd_ram 的位址為 0x0101
```

注意：

在上述的例子中，運算式++lcd_ram; 或 lcd_ram = lcd_ram+1;，因 TM89 C 編譯器以 8 位元為指標位移單位，這就是為何 lcd_ram 的位址會變成是 0x0102（即 0x0100 位移 1 byte = 0x0100 位移 2 nibble = 0x0100 + 0x0002 = 0x0102）。

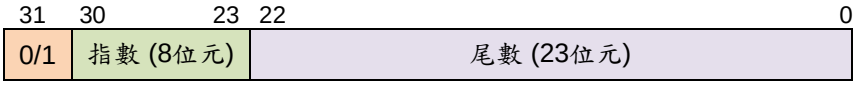
浮點數（float）

在計算中，浮點數（floating point）引進了一種代表實數的方式，其可支援一較大數值表示。在一般情況下，數值的近似值由一固定數值的符號數字和指數部份運算而得。典型的數字表示的形式為：

$$\text{Significant digits} \times \text{base}^{\text{exponent}}$$

TM89 C 編譯器支援之浮點數，為符合 IEEE-754 單精度浮點數的標準，即佔 32-位元（4 位元組）記憶體空間的二進位表示式。

IEEE754 單精度浮點數的格式（32 位元）：



- 第 31 個位元：正、負號（sign bit） - 0 表正數， 1 表負數。
- 第 23-30 位元：指數（exponent field），以超 127 表示（excess-127）。
- 最後 0-22 位元：尾數（mantissa field），為經過正規化後的資料。

範例：

```
float f_value = 23.123;
```

您依然可用科學記號表示法來表示一較大的數值，如下例所示。

```
Float f_value = 4.0e6;
```

或

```
float f_value = 4.0E6;

// 4,000,000 = 4.0 * 106 = 4.0e6 //
```

結構和聯合宣告

結構 (Structure) 為一個物件其包含一群有序的資料成員。不同於矩陣的元素，結構裡面的成員可以有多種不同的資料型態。**聯合 (Union)** 為一個可以在指定的時間內包含任何一個成員的物件。結構和聯合具有同樣的型態，其語法如下：

struct-or-union-specifier:	*struct-or-union {struct-decl-list}*
	struct-or-union identifier {struct-decl-list}
	struct-or-union identifier
struct-or-union:	**struct**
	union

struct-decl-list 為結構或聯合中一系列的成員宣告。

可利用結構來聚集具邏輯相關性的物件。在以下範例，從程式行 int street_no; 到 char *postal_code; 宣告了 structure tag 的位址：

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};

struct address perm_address;
struct address temp_address;
struct address *p_perm_address = &perm_address;

/*
The variables perm_address and temp_address are instances of the structure data type address.
Both contain the members described in the declaration of address. The pointer p_perm_address
points to a structure of address and is initialized to point to perm_address.
*/
```

列舉宣告

列舉是一資料型態其包含一組稱為常整數的值。其語法如下：

enum-specifier:	**enum** *{enum-list}* **enum** *{identifier enum-list}* **enum** *identifier*
enum-list:	*Enumerator* *enum-list , enumerator*
enumerator:	*Identifier* *identifier = constant-expression*

當您定義一個列舉的資料型態，您定義一組識別符號。識別符號以**常整數**宣告並且可出現在任何這常數被允許的地方。在這集合裡的每一個識別符號為列舉常數（***enumeration constant***）。

列舉常數的值是由以下方式決定：

1. 經由賦值符號（=）和常數表示式，明確定義列舉常數一常數的值。
2. 若沒有給定一明確的值，列表最左邊的常數預設為零（0）。
3. 識別符號沒有明確定義數值，直接被分配比前一個識別符號的值增加一。

範例：

```
enum grain { oats, wheat, barley, corn, rice };
             /*   0      1       2        3     4      */
enum grain { oats=1, wheat, barley, corn, rice };
             /*   1        2       3        4      5   */
enum grain { oats, wheat=10, barley, corn=20, rice };
            /*   0        10       11       20        21   */
```

型態限定詞（Type Qualifiers）

型態限定詞的語法：

type-qualifier:	**const**

const 限定詞明確宣告一個資料物件為無法再被變更的資料項目。您無法在一個可變更左值（lvalue）的運算式中使用 const 資料物件。例如，const 資料物件不可出現在賦值敘述的左邊。雖然 const 變數不可更改，仍可按照初始化物件的規則來給定初始值。

在 TM89 系列 C 語言編譯器，全域常數變數為了節省 RAM 記憶空間，將被定址到程式 ROM 的記憶體中。區域常數變數仍被指向 RAM 的記憶體。

範例：

```
const char _szmydata[] = ”hello”;
const unsigned char _szdata[]={0x10,0x20,0x30,0x40,0x50,0x60};
const int _idata=0x55AA;
cosnt int *pointer;          // const pointer
```

在使用**常數指標**（const pointer）時，需注意下列幾點：

1. 常數指標變數必須只能宣告成指向常數或陣列型態變數的指標。
2. 宣告在全域（global）區域的常數指標
   A. 將定址到 Table ROM 的記憶體空間。
   B. 以 Byte 為定址單位。
3. 宣告在區域（local）處的常數指標
   A. 將定址在 Data RAM 的記憶體定間。
   B. 以 nibble 為定址單位。
4. 因為全域與區域常數指標的定址記憶體定間不同，建議區域與全域常數指標變數不要有相互指派的情況，以免產生不可預期的錯誤。
5. 在初始化全域常數指標位址時， 如下例，全域常數指標 p 指向一常數矩陣啟始位址時，建議在程式應用處才實際賦予指標 p 所指之位址。

例如：

```
const int array[5] = {1,2,3,4,5};
const int *p = array;                    // 不建議
 main ()
{
      ++p;
}
```

建議修改方式：

```
const int array[5] = {1,2,3,4,5};
const int *p;
main()
{
      p=array;        //建議在實際應用時才指定位址
      ++p;
}
```

宣告（Declarators）

宣告 指出宣告式中資料物件或函式的作用域、儲存持續時間和型態。每個識別字宣告只包含唯一的識別字。在宣告"**T D1;**" **D1** 為一識別字，其資料型態為 **T**。

在一個宣告中，您可以指明一個物件的資料型態為矩陣、指標或一個參考點（reference）。您也可以在宣告中作初始化的動作。以下表格說明一些宣告的範例：

範例	說明
int year	year為一個 **整數** 資料物件
int *node	node為指向 **整數** 資料物件的指標
int name[126]	name為含126個 **整數** 元件的矩陣
int *move( )	move為回傳一**整數指標**的函式
extern const int sys_clock	sys_clock為一個**常整數**而且為外部鏈結

TM89 C 語言編譯器作用於 4-位元的單晶片，其有定義與支援的資料型態長度如下列表：

資料型態	大小 (bytes)	範圍
char	1	-128 ~ 127
unsigned char	1	0 ~ 255
short	2	-32768 ~ 32767
unsigned short	2	0 ~ 65535
int	2	-32768 ~ 32767
unsigned int	2	0 ~ 65535
long	4	-2147483648 ~ 2147483647
unsigned long	4	0 ~ 4294967295
pointer	2	0 ~ 65535
nibble	4	-8 ~ 7
unsigned nibble	4	0 ~ 15
float	4	≈± 1.2 x 10^{-38} ~ ≈±3.4 x 10^{38}

指標宣告（Pointer Declarators）

指標型態變數內含值為物件或函式的位址。指標可以指向任何一個資料型態的物件除了參考（reference）。指標被歸類為數量（scalar）型態，意思指每次只能擁有一個值。指標宣告的格式如下：

pointer:	** type-qualifier-listopt*
	** type-qualifier-listopt pointer*

當您在一個賦值運算式中使用指標，您必須確保運算裡的指標型態是可相容的（compatible）。意思是指兩個指標型態有相同的型態限定詞並為可相容，則他們指向可相容型態的物件。

範例：

```
int section[80];
int *student = section;
```

一些常用的指標用法如下：

- 存取矩陣的元素或結構的成員
- 存取一字元矩陣如同存取一個字串
- 傳變數的位址到一個函式。透過變數的位址，函式可參考到該變數，並變更其變數內容值。

矩陣宣告（Array Declarators）

矩陣是擁有同樣資料型態的物件聚集。矩陣裡面的單獨物件，稱為元素，可從其在矩陣中的位置來存取。下標操作符（[ ]）提供一讓矩陣元素擁有一索引的機制。

array:	*Type identifier [constant-expression]*

矩陣的初始化是以包含在括弧裡（{ }）的以逗號隔開之常數表示式列單來初始化。初始值前面是一個賦值符號（=）。

範例：

```
int number[3] = { 5, 7, 2 };
```

矩陣數字包含以下值：number[0]為 5；number[1]為 7；number[2]為 2。

範例：

```
int item[ ] = { 1, 2, 3, 4, 5 };
```

因為沒有明確說明其大小，TM89 C 語言編譯器給定五個初始值的元素，而且包含五個初始值。

初始化一個字串固定會擺放一個空字元（\0）在字串的結尾（若空間足夠或矩陣大小沒有事先定義）。以下定義為字元矩陣的初始化：

```
static char name1[ ] = { 'J', 'o', 'y' };
static char name2[ ] = { "Joy" };
static char name3[4] = "Joy";
```

以下定義明確初始化 12 元素矩陣當中的六個元素：

```
static int matrix[3][4] =
{
      {1, 2},
      {3, 4},
      {5, 6}
};
```

元素	數值	元素	數值	元素	數值
matrix[0][0]	1	matrix[1][0]	3	matrix[2][0]	5
matrix[0][1]	2	matrix[1][1]	4	matrix[2][1]	6
matrix[0][2]	0	matrix[1][2]	0	matrix[2][2]	0
matrix[0][3]	0	matrix[1][3]	0	matrix[2][3]	0

以下規定應用在矩陣宣告：

- 若矩陣為固定長度的矩陣，矩陣大小表示式是用方括弧包住。若表示式確定存在，其值是一個大於零的整數值。
- 當幾個定義的矩陣為相鄰，意思是指他是一個多元矩陣；定義矩陣的範圍的常數表示式只能少掉序列中的第一個成員。
- 若此矩陣由外部且為確實的定義（其定義足以用來配置儲存空間），或者在宣告後再作初始化，則矩陣的第一個維度可忽略不宣告大小。後者的情況，矩陣大小是從提供的元素數量算出。
- 若矩陣大小表示式為常整數運算式，則矩陣型態為“固定長度矩陣”，並且元素類別有固定大小。
- 為了讓兩個矩陣型態可相容，其元素型態必須具相容性。再者，若二者的大小表示式為常整數運算式，則他們的結果值必須為相等的值。

函式宣告和原型

當呼叫一個函式，其函式原型是在作用域時，程式將轉換每一個實際參數（actual parameter）的型態為相對應在函式原型中宣告的正式參數（formal parameter） 之資料型態，以取代預設的參數。出現在函式參數列表中的參數數量一定要與函式原型的數量吻合。

以下為兩個函式原型的範例：

```
long foo(int *first, int second);
int *fip(int a, long l, int b);
```

建議使用 ANSI C 函式原型宣告。在傳統 C 語言，原型的應用未完整。原型的應用在 ANSI C 和傳統 C 仍存在著很明顯的差異。

ANSI C	傳統 C
void adjust_xy (short x, short y) {…..}	void adjust_xy (x, y) short x; short y; {…..}

注意：

- **不支援“遞迴函式”（recursive function ）**
- 函式原型在函式呼叫前必須先宣告

asm 宣告

關鍵字 **asm** 指的是組合語言（assembly code）。在實行的過程中，TM89 系列 C 語言編譯器認得並且忽略宣告式的關鍵字 asm。其文法如下：

```
asm (<string literal>[, optional parameters]) ;
or
__asm__ (<string literal>[, optional parameters]) ;
```

asm 語句只可用於函式裡（勿用於全域區，global area）。內部組合語言式子為主要的式子，所以也可以用在表示式子的一部分。請注意，一個含在內部組合語言的運算式，其運算結果的資料型態永遠為 void。

字串內容將由編譯器先行解讀，並且加入到所產生之組語以輸出，這樣才可以被後端尤其是優化器進行處理。因此，編譯器只允許正規的 TM89XX 操作碼（opcodes）用在內部組譯器。助憶指令（pseudo）（如.import，.byte…等等）是不被接受的，即使在 cc89 組譯器（用來編譯為組合語言程式碼）是被接受的。

內建的內部組譯器並非取代和編譯器一起的完善的巨集組譯器。

注意：內部組譯器式子為編譯器產生的最佳化結果。目前沒有方法可以保護內部組譯器式子不受優化器移動或移除。若有疑慮，可檢查產生的組合語言輸出或取消優化選項。

字串可以包含以下所表列的格式符號。在傳送組合語言程式碼到後端處理之前，運算式子可插入一些格式符號。

格式符號	說明
%b	8位元數字值
%w	16位元數字值
%l	32位元數字值
%v	（全域）參數或函式的組譯器名稱
%o	（區域）參數的堆疊偏移量
%%	% 符號

使用這些符號，您可以存取 C 語言的 #define，參數或從內建組譯器類似的東西。例如，把 C 語言 #define 的值載入@HL 索引暫存器，可用以下寫法：

```
#define OFFS  23
asm__ ("LDS %b", OFFS);
```

或，要存取靜態變數的結構成員：

```
#define offsetof(type, member)  (unsigned) (&((type*) 0)->member)
typedef struct {
   unsigned char x;
   unsigned char y;
   unsigned char color;
} pixel_t;
static pixel_t pixel;
_asm__ ("SHLX");
__asm__ ("SETDAT %v+%b", pixel,offsetof(pixel_t, color));
asm(“LDS8 @HL,%b”,OFFS);
```

注意：請勿把組譯器標籤用於全域參數的名稱或函式併入您 asm 式子。

如以下例子：

```
int foo;
int bar () { return 1; }
void main()
{
      __asm__ ("lda _foo");   /* 請勿這麼作! */
      ...
      __asm__ ("call _bar");   /* 請勿這麼作! */
}
```

全域變數及函式名稱在經過 c 編譯器編譯過後產生的 .s 檔中，原名稱將變成以 ’ _’ 為前導的變數或函數名稱。 如上例之全域變數宣告: int foo=0; 在相對應轉出的 .s 檔中為以下內容：

```
SHLX
SETDAT _foo+0
LDS8# @HL,$00
LDS8# @HL,$00
```

變數名稱 foo 已轉為 _foo；請注意，對於區域變數或參數名稱…等，並非依循此名稱轉換規則。故而，為避免變數、參數或函式名稱在 asm 式子中的併字與維護問題。在 C 程式中，使用 inline asm 式子時，建議使用格式符號來替換變數名稱，用法如 __asm__("SETDAT %o",foo);。同樣地，在呼叫無需參數傳入的函式時， 如 __asm__(“call _bar”), 建議直接以 bar(); 替代。

範例 1：在 C 函式中撰寫全組合語言的程式：

```
char* strcpy(char *dest,char *src)
{
      asm("SZRX");
      asm("SETDAT  %o",src);// // Set offset of LOCAL name src
      asm("SHLX");
      asm("SETDAT $0080");
      asm("LID8$   @HL,@ZR");
      asm("LID8$   @HL,@ZR");
      asm("SZRX");
      asm("SETDAT  %o",dest); // Set offset of LOCAL name dest
      asm("SHLX");
      asm("SETDAT $00C0");
      asm("LID8$   @HL,@ZR");
      asm("LID8$   @HL,@ZR");
      asm(" MHL 0");
      asm(" MZR 0");
      asm("L1:"); // generate label name L1
      asm("LID8%%   @ZR,@HL");
      asm("LDA#  @HL");
      asm("OR#  @HL");
      asm("JZ EXIT");
      asm("JMP L1");
      asm("EXIT :"); // generate label name EXIT
      asm("SHLX");
      asm("SETDAT  %o",dest); //  return dest pointer in HL index.
      asm("RTS");
}
```

範例 2：在*.C 檔案呼叫 ASM

```
main(void)
{
   asm(“CALL  asmLabelDelay”);        //in *.ASM file need “.export asmLabelDelay”
}
```

宣告的限制

並非所有可能的宣告語法都可以使用。其用法有以下限制：

- 雖然可以回傳指標，函式無法回傳矩陣或函式。我們建議使用以下例子回傳函式的矩陣：

```
int* subFoo(int x);
void main(void)
{
    ……
    int C[10],*p;
    p = subFoo(value1);
    for(i=0; i<10; ++i)
    {
        C[i] = *p;
        ++p;
    }
    ….
}
// subFoo: assign array elements
int* subFoo(int x)
{
   int B[10];
   int i;
   for(i=0; i<10; ++i)
     B[i] = 10;
   return B;
}
```

- 雖然指標指向函式的矩陣可以存在，可是函式的矩陣是不存在的。
- 結構或聯合不能包含一個函式。以下結構宣告的例子是不合規定的：

```
struct ERROR_STRUCT
{
   int i;
   int y;

   int foo(int var)
   {
      .....
      return var;
   };
};
```

型別定義（Typedef）

typedef 宣告讓您定義您自訂的識別式子，並且可以用來代替型態識別字如 int、long、struct 和 pointer。使用儲存類別 typedef 宣告並沒有預留其空間。您用 typedef 定義的名稱並不是新的資料型態，而是資料型態的同義或他們代表的資料型態的組合。

以下式子宣告 TLENGTH 為同義於 int 並且用 typedef 來宣告 length、width 和 height 為整數參數：

```
typedef int TLENGTH;
LENGTH length, width, height;
```

以下宣告相當於上面的宣告：

```
Int length, width, height;
```

同樣的，typedef 可用來定義物件類別例如 struct 和 union。

範例：

```
typedef struct {
    int scruples;
    int drams;
    int grains;
} WEIGHT;
```

結構 WEIGHT 可用在以下宣告：

```
WEIGHT chicken, cow, horse, whale;
```

初始化（Initialization）

物件或未知大小的矩陣的宣告可用來描述宣告式中識別符號的初始值。初始化前面為 '='並包含一個式子或以括弧包住的一系列數值：

initializer:	*assignment-expression*
	{initializer-list}
initializer-list:	*Initializer*
	initializer-list , initializer

聚集的初始化（Initialization of Aggregates）

在 TM89 C 語言編譯器，*struct* 或 *union* 類別的物件可被初始化，即使他們有自動儲存持續期間。*union* 是用第一個宣告的元素的類別來初始化。當參數被宣告為 *struct* 或 *array*，初始化包含一個由括弧包圍住並以逗號隔開的初始列，來對以遞增的下標符號或依成員的順序之聚集的成員做初始化。

範例：

```
union dc_u {
    int  d;
    char *cptr;
};
union dc_u dc0 = { 4 };
```

最後的縮寫允許字元矩陣初始化為字串。在此狀況，字串中連續的字元依序為矩陣的成員做初始化。

```
char msg[] = "Syntax error on line %s\n";
```

4. 算式和運算符號（Expressions and Operators）

此章節介紹 C 語言可用的各種運算式和運算子。此章節說明的運算式和運算子大約按照其優先序。

在 C 語言的運算符號優先序和結合律規則

C 語言的運算子有優先序和結合律規則，其用以決定運算式如何將運算子做組合且算出結果。優先序是指對不同類型的運算子和運算式做組合之優先權。結合律決定擁有同樣優先權的運算子和運算式，其 "由左至右" 或 "由右至左" 的結合順序。運算子的優先權是具意義的，尤其在當有其他具更高或更低優先樣的運算子同時存在時。運算式中具更高優先權的運算子將被優先處理。

以下表格按照其優先律高低列出 C 語言運算子，並且顯示每個運算子的結合律的方向（L-R 指 "由左至右"，R-L 指 "由右至左"）：

Tokens（優先順序由高至低）	運算子	類別	結合律
識別符號，常數，字串，括弧的式子	主要式子	主要	
() [] -> .	函式呼叫，下標符號，間接選項，直接選項	後置	L-R
++ --	增加，減少（後置）	後置	L-R
++ --	增加，減少（前置）	前置	R-L
! ~ + - & sizeof *	邏輯和逐位NOT，單元素的加和減，位子，大小，間接	單元素	R-L
(type)	強制轉型	單元素	R-L
* / %	倍加的	雙元素	L-R
+ -	加法的	雙元素	L-R
<< >>	左移，右移	雙元素	L-R
< <= > >=	關係比較	雙元素	L-R
== !=	等式比較	雙元素	L-R
&	位元"和"	雙元素	L-R
^	位元異算符	雙元素	L-R
|	逐位"或"算符	雙元素	L-R
&&	邏輯"和"算符	雙元素	L-R
||	邏輯"或"算符	雙元素	L-R
? :	條件式	三元素	R-L
= += -= *= /= %= ^= &= |= <<= >>=	分配	雙元素	R-L
,	逗號	雙元素	L-R

主要的式子（Primary Expressions）

以下皆列為“主要的式子”：

識別符號	識別符號意指物件的左值（lvalue）。識別符號指向一函式，則為一個函式指示符
常數	常數的資料型態是由其格式和數值來決定
字串	字串類型是字元的矩陣，藉由下標符號做修改
以括弧包住的式子	有括弧的式子和值與不括弧的式子為相同的。括弧符號的存在不影響運算式為左值（lvalue）、右值（rvalue）或函式指示符 的式子

後置式（Postfix Expressions）

後置運算子為在運算元後面出現的運算子。後置運算式為主要式子，或包含後置運算子的主要式子。以下總結可用的後置運算子：

運算子功能	用法	範例
member selection	object . member	Table.Color
member selection	pointer -> member	Table->Color
Subscripting	pointer [expr]	ArrayOne[2]
function call	expr (expr_list)	Foo(a,b,c)
value construction	type (expr_list)	Long(intOne)
postfix increment	lvalue --	Lindex--
postfix decrement	lvalue ++	Lindex++

矩陣下標符號運算子（Array Subscripting Operator）

矩陣的元素以一個後置運算式後面跟著方括弧（[ ]）的式子來表示。括弧裡面的句子當作下標符號參考。矩陣第一個元素的下標符號為 0。句子 code[10]指的是矩陣的第 11 個元素。

在多維矩陣，您最常用的方式是以增加最右邊下標符號來參照每一個元素（依照逐增的儲存位置）。例如，以下句子為每一個矩陣 code[4][3][6]元素給予數值 100：

```
Int first, second, third;
for (first = 0; first < 4; ++first)
{
    for (second = 0; second < 3; ++second)
    {
        for (third = 0; third < 6; ++third)
        {
            code[first][second][third] = 100;
        }
    }
}
```

結構和聯合的參照（Structure and Union References）

結構或聯合的參照是以一個後置運算式後面跟著一個點 (.) 和識別符號來表示。其語法如下：

postfix-expression. identifier

postfix expression 必須為一個結構或聯合，而且 *identifier* 必須為結構或聯合的成員。其值為結構或聯合的成員的值，而且若第一個句子為左值 (lvalue)，他也必須為左值。運算結果含有結構或聯合指定成員的型態與結構或聯合的限定詞。

間接結構和聯合參照（Indirect Structure and Union References）

透過 ->（箭頭）運算子來存取結構或聯合的成員，意即使用指標。一個間接結構或聯合的參照表示式為，後置運算式後面跟著一個箭頭（由 - 再加 >）和識別符號。其語法如下：

postfix-expression-> identifier

postfix expression 必須為指向一個結構或聯合的指標，而且 *identifier* 必須為結構或聯合的成員名稱。結果是左值，意指是參考後置運算式所指向的結構或聯合之成員。其運算結果含有結構或聯合指定成員的型態與結構或聯合的限定詞。故，式子 E1->MOS 相等於 (*E1).MOS。

後置++和後置--（Postfix ++ and Postfix --）

後置++和後置--的語法如下：

postfix-expression ++
postfix-expression --

當後置 ++ 應用到可更改的左值（lvalue），其結果是左值所參照的物件值。當其結果被指明後，物件將被增加 1。其結果的資料型態相同於左值運算式的資料型態。他的結果並非一個左值。

當後置 -- 應用到可更改的左值（lvalue），其結果是左值所參照的物件值。當其結果被指明後，物件將被減 1。其結果的資料型態相同於左值運算式的資料型態。他的結果並非一個左值。

後置 ++ 和 -- 的運算子，運算式的儲存值的更新將被延誤到下一個序列點。

單元運算子（Unary Operators）

單元運算式包含一個運算元和一個單元運算子。所有單元運算子具有相同優先權與***由右至左的結合律***。因此單元運算子為後置運算式。如下面的說明，通常的算數轉換是在單元運算式的運算元中進行。以下表格總結單元式子的運算符號：

運算符號功能	用法
物件的大小（bytes）	**sizeof** (expr)
類別大小（bytes）	**sizeof** type
前置遞增	++ lvalue
前置遞減	-- lvalue
補數	~ expr
非	! expr
單元負號	- expr
單元正號	+ expr
位址	& lvalue
間接引用或反參照	* expr

位址或間接引用運算子（Address-of and Indirection Operators）

單元運算子 * 意指間接引用；型態轉換式必須是一個指標，而且其結果會是運算式指向物件的左值或者一個函式的指定符。單元運算子 & 的運算元可以是函式指定符或是指向物件的左值。單元運算子 & 的運算結果是依左值，來指向一物件的指標或是依函式指定符參考一函式。

單元運算子 + 和 -（Unary + and Unary – Operators）

單元運算子 - 的結果值是運算元的負數。整數四捨五入會在運算式進行而且其結果會是四捨五入後的類別和運算式的負數值。

單元運算子 + 維持運算式的值。運算元可以是算數的任何類別或指標類別。其結果並非是一左值。

邏輯否定！和位元否定 ~ 運算子

邏輯否定運算子 ！ 決定運算元運算結果為 0（假）或非零（真）。邏輯否定運算子！的結果為 1 若運算元的值為 0，或結果為 0 若運算元的值為非零。

以下兩個式子的運算結果是相同的：

```
!right;
right == 0;
```

位元否定運算子 ~ 產生運算元的位元方式之補數。結果以二進制表示，在二進制運算式的表示法中每個位元會是相同位元的相反值。運算元必須為整數類別。其結果值的型態與運算式的型態相同且不為左值。

例如：

設 x 代表十進制數值 5。以 4 位元二進制表示 x 為：0101。式子~x 產生的結果是： 1010

前置 ++ 和 -- 運算子

前置運算子 ++ 和 -- ，遞增和遞減其運算元。其語法如下：

```
++unary-expression
--unary-expression
```

由可變動左值前置運算子 ++ 所指向的物件，遞增其值。運算式的值為運算元的新值但非左值。運算式++x 與 x += 1 是相同的。前置 -- 遞減其左值運算元的方式如同前置 ++遞增的作法。

sizeof 單元運算子

sizeof 運算子得到運算式的大小，以 hibble 為單位。其可以是型態的運算式或有括弧的型態名稱。

在 TM89 C 語言編譯器，一個字元 *char* 的大小為 2，一個整數 *int* 的大小為 4，一個長整數 *long* 的大小為 8，一個浮點數 float 的大小為 8。其主要用在程序的溝通，例如配置儲存體和 I/O 系統。sizeof 運算子的語法如下：

```
sizeof unary-expression
sizeof (type-name)
```

sizeof 運算子**不能**應用於：

- 函式類別
- 未定義的結構或類別
- 不完整的類別（如 void）

乘法運算子（Multiplicative Operators）

乘法運算子 *、/ 和 % 其結合性為“由左至右”。此運算子將執行一般的算術轉換。以下是乘法運算子的語法：

multiplicative expression:	*cast-expression* *multiplicative-expression * cast-expression* *multiplicative-expression / cast-expression* *multiplicative-expression % cast-expression*

* 和 / 的運算元必須為算術類別。% 的運算元必須為整數型態。二元運算子 * 意指乘法，其結果是運算元的乘積。二元運算子 / 意指第一個運算元（被除數）除於第二個運算元（除數）。整數相除的結果為整數，商數其大小少於或等於正商數而且具相同的正負號。

二元運算子 % 的結果值為第一個運算元（被除數）除於第二個運算元（除數）的餘數。運算元必須為整數。

加法運算子（Additive Operators）

加法運算子 + 和 - 的結合性為“由左至右”。此運算子將執行一般的算術轉換。以下是加法的運算子的語法：

additive-expression:	*multiplicative-expression* *additive-expression + multiplicative-expression* *additive-expression - multiplicative-expression*

指向矩陣物件的指標可與一整數型態的值相加。其結果為型態與運算元指標相同的指標。其結果參照到矩陣的其他元素，其下標值為原本元素向後位移，其位移量為其所加的整數數值。若指標結果指向矩陣以外的儲存空間，而非矩陣以外的第一個位置，其結果是不被定義的。編譯器不提供指標的合法範圍檢查。例如，在做加法後，ptr 指標指向矩陣的第三個元素：

```
int array[5];
int *ptr;
ptr = array+ 2;
```

位移運算子（Shift Operators）

位元方式位移運算子移動二進制物件的位元值。位元方式位移運算子 << 和 >> 具“由左至右”結合性。每個運算元必須為整數型態。整數四捨五入都有應用到每個運算式。其語法如下：

shift-expression:	*additive-expression* *shift-expression << additive-expression* *shift-expression >> additive-expression*

運算子	功能
<<	表示位元將被位移至左邊
>>	表示位元將被位移至右邊

例如，若 left_op 值為 4019，其位元樣式（以 16 位元格式）為：

0000111110110011

式子 left_op << 3 結果為：

0111110110011000

關係運算子（< > <= >=）

關係運算子對兩個運算元作比較並且決定一個關係的正確性。其結果的型態為 int 而且若數值為 1 則定義關係為真（true）或 0 則關係為假（false）。其結果並非為左值。

運算子	功能
<	表示左邊的運算元是否小於右邊的運算元
>	表示左邊的運算元是否大於右邊的運算元
<=	表示左邊的運算元是否小於或等於右邊的運算元
>=	表示左邊的運算元是否大於或等於右邊的運算元

當運算元為指標，其結果由指標參照的物件的位置來決定。若指標不是參照到相同矩陣的物件，結果會是未定義的。若指標參照到相同的物件，他們會被視為相同的。

相等運算子（== !=）

如同關係運算子，相等運算子為比較兩個運算元來決定其關係的正確性。然而，相等運算子的優先權低於關係運算子。其結果的型態為 int 而且數值為 1 若定義的關係為真（true），或 0 若關係為假（false）。

運算符號	功能
==	表示左邊運算元的值相等於右邊運算元的值
!=	表示左邊運算元的值不等於右邊運算元的值

邏輯運算子 AND（&&），邏輯運算子 OR（||）

邏輯運算子 AND（&&）判斷兩者運算元是否皆為真（true）。若兩者運算元皆為非零的值，其結果為 1。反之，其結果為 0。運算結果的型態為 int。兩者運算元必須皆為算數或指標型態。邏輯 AND 也有對每個運算元進行一般的算術轉換。

句子	結果
1 && 0	0
1 && 6	1
0 && 0	0

邏輯運算子 OR（||）表示其中運算元是否為真（true）。若其中之一運算元為非零的值，其結果為 1。反之，其結果為 0。運算結果的型態為 int。兩者運算元必須擁有算數或指標型態。邏輯 OR 也有對每個運算元進行一般的算術轉換。

句子	結果
1 || 0	1
1 || 6	1
0 || 0	0

條件運算子（Conditional Operator）

條件運算式是一個複合運算式，其包含一個轉換成 bool（$operand_1$）的條件，計算並判斷運算式的值，若值為真則計算 $operand_2$，若值為假則計算 $operand_3$。

($operand_1$? $operand_2$: $operand_3$)

計算第一個運算元以其結果的數值，決定是否要計算第二或第三運算元：

- 若值為真（true），第二個運算元將被處理
- 若值為假（false），第三個運算元將被處理

條件式的結果為第二或第三運算元的值。

以下句子決定 y 或 z 哪一個參數為比較大的值，，並且把比較大的值指向參數 x：

```
x = (y > z) ? y : z;
```

以下為相同意義的句子：

```
if (y> z)
    x = y ;
else
    x = z;
```

5. 述句（Statements）

述句是給電腦讀的一完整的指令，其為最小的獨立計算單元，其明確地說明將進行的動作。在大多數狀況下，述句為依序執行。

算式述句（Expression Statements）

通常算式述句是對運算式計算其副效應（side effect），例如賦值或函式呼叫。有一個特例是空述句，只包含一個分號。

算式的範例：

```
marks = dollars * exch_rate;              /* 指派給 marks */
(difference < 0) ? ++losses : ++gain;     /* 條件式遞增 */
```

區塊述句（Block Statement）

區塊述句、或複合述句，讓您聚集任何數據定義、宣告、和述句，以成為一個述句。在複合述句的宣告是具區域的作用域（block scope）。若在宣告列中的識別符號曾被宣告過，外部宣告將被隱藏於此區塊的期間，直到它獲得焦點後才恢復。

選擇述句（Selection Statements）

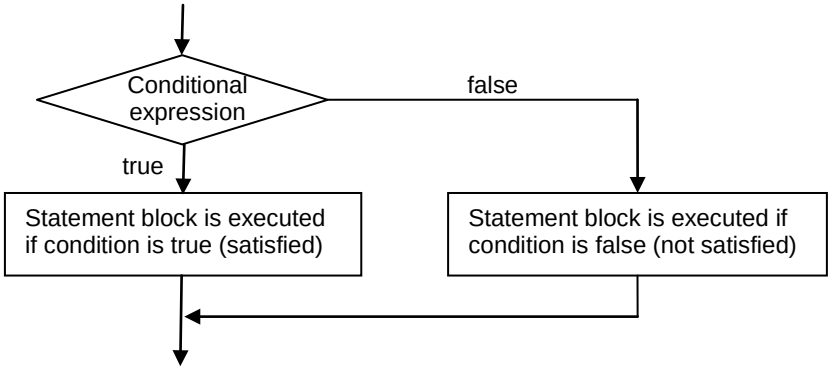
選擇述句包含 if 和 switch 述句。選擇述句將根據算式的判斷，以選擇其中一項述句來執行。***Expression*** 為控制運算式。

selection-statement:	**if** *(expression) statement* **if** *(expression) statement* **else** *statement* **switch** *(expression) statement*

if 述句

if 述句是一個選擇述句，其允許有多種可能性的控制流程。您可以在 if 述句裡面，選擇性的指定一個 else 子句。若測試算式結果為 0 而且存在 else 子句，述句將執行 else 子句。若測試算式結果為非零值，述句將執行算式並且忽略 else 子句。

當 if 述句是巢狀的並且存在 else 子句，給定的 else 子句將與同一個區塊中最相近的 if 述句結合。



switch 述句

switch 述句是一個選擇述句，讓您可以根據 switch 述句的值，來控制轉移到不同的述句。switch 述句必須為對一個整數或序列值做計算。switch 述句的內容包含 case 子句如：

- case 標籤
- 選擇性的 **default** 標籤
- 一個 **case** 算式
- 一序列的述句

若 switch 算式值與一個 case 的算式值符合，跟在 case 算式後面的述句將被執行，直到遇到 break 述句或到了 switch 內容的終點。

範例：

```
char key;
…..
switch (key)
{
   case '+':
        add();
   break;
   case '-':
        subtract();
        break;
   case '*':
        multiply();
        break;
   case '/':
        divide();
        break;
   default:
       break;
}
```

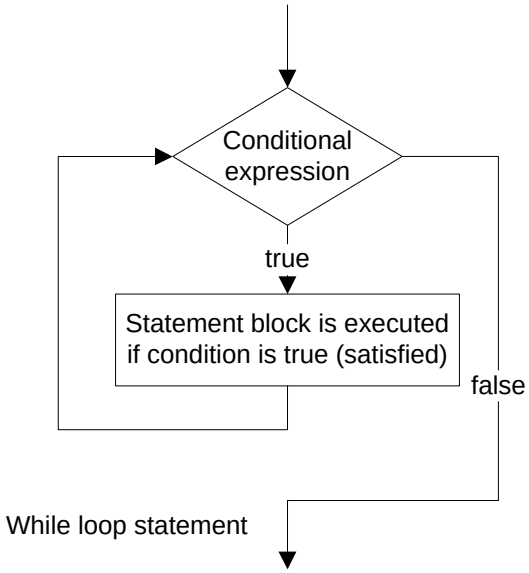
重複述句

重複述句重複執行附屬的述句（稱為 body），直到控制算式為 0。在 for 述句，第二個算式為控制算式。其格式如下：

iteration-statement:	**while** (*expression*) *statement* **do** *statement* **while** (*expression*) ; **for** ([expression] ; [expression] ; [expression]) statement

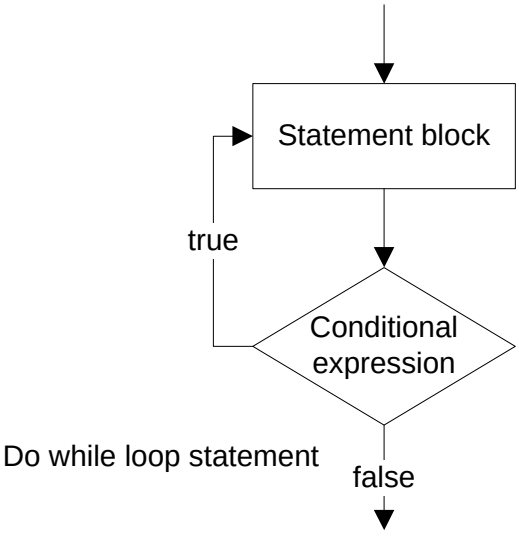
while 述句

while 述句重複執行迴圈裡的 body 直到控制算式為 0。一個 break、return、或 goto 述句可讓 while 述句終止，即使控制算式不是為 0。



do 述句

不同於 while 述句，do-while 述句的控制算式是在執行過一次 body 後才判斷。因為執行的順序，所以述句至少會被執行一次。



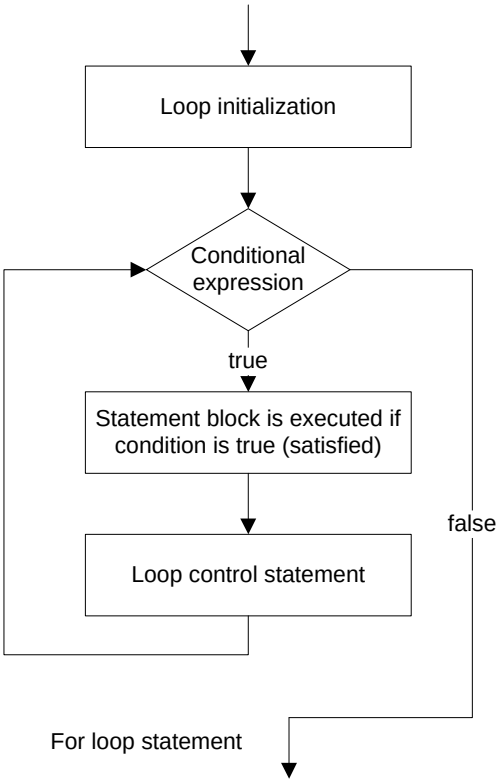
for 述句

for 述句讓您作以下的事：

- 在執行第一個述句前，先判斷述句（初始化）。
- 指明一運算式以判斷述句是否該進行（依條件）。
- 在每次述句的迭代後，執行運算式（常用來遞增每次的重複）。
- 若控制算式不是 0，則重複處理述句。

break、**return** 或 **goto** 述句可讓 for 述句終止，即便第二個算式不是為 0。若您省略 $expression_2$，您必須要用 **break**、**return** 或 **goto** 述句來終止 for 述句。

第一個算式說明迴圈的初始化。第二個算式為控制算式，在每次重複之前會被判斷。第三個算式通常說明其遞增狀況。這是在每次重複後被檢查的。



jump 述句

jump 述句將造成無條件式的控制轉換，其語法如下：

jump-statement:	**goto** *identifier*; **continue;** **break;** **return** [*expression*]

goto 述句

goto 述句讓您的程式無條件地被轉到 goto 述句所指定的標籤（label）述句。

goto *identifier;*

識別符號必須命名為處於封閉式函式內的標籤。若標籤名稱未出現，將被視為隱藏式的宣告。因為 goto 述句會干擾正常的流程序列，造成程式變得更難閱讀和維護。通常，break 述句，continue 述句，或函式呼叫可以取代 goto 述句的需求。

範例：

```
int i=0;
Label:
if( i < 10 )
   goto Label;
```

continue 述句

continue 述句只能出現在重複述句的 body。continue 述句會導致控制略過循環的後續部分到達循環的端點，即最小的封閉 while,do, for 述句。

```
while (condition)
{
……
     if(…)
     continue;
……
……
}
```

break 述句

break 述句讓您終止一個反覆的述句（**do**，**for**，或 **while**）或一個 **switch** 述句，而且從任何非邏輯終點的地方跳出來。一個 break 只能出現在這些述句之一。在反覆述句中，break 述句終止迴圈並且移動控制到迴圈以外的下一個述句。

```
while (condition)
{
……
……
     break;
……
……
}
```

return 述句

return 述句終止目前執行的函式並且回傳控制到函式的呼叫者。若目前的函式型態為 void，則 return 述句不可以是算式。若尚未執行到明確的 return 述句，而先到達函式的終點，將執行隱含式的 return（不含算式）。

標籤述句

標籤有三種：**identifier**、**case** 和 **default**。標籤述句語法如下：

Labeled-statement:	***identifier*** *: statement* **case** *constant-expression : statement* **default** *: statement*

任何述句可以為一個標籤附加一個簡單的識別符號。此標籤的作用域為目前的函式。所以，標籤名稱在函式內必須為唯一的。

中斷

與一般用途的電腦相比，應用在嵌入式系統的微型控制器，通常透過指令處理能力以尋求中斷延遲的最佳化。一些問題包含降低延遲，讓他變成更可以預測（支援及時控制）。十速科技微型控制器提供及時的（可預測，雖然不見得很快）反應到嵌入式系統內控制的事件。中斷應用將產生 timer 和 counter 的直接改變。在 TM89 C 語言編譯器，中斷的格式如下：

Interrupt service routine :	void interrupt <function(void)> @ <interrupt vector address>

<interrupt vector address> 指很多在 MCU 的中斷向量，在 TM89 系列晶片中，會有以下這些中斷函式：5 個外部中斷因子 （INT pin, Port IOA, IOC, IOD & KI input），及 5 個內部中斷因子（Pre-Divider, Timer1, Timer2, Timer3 & RFC）。我們提供序列 0x10(INT)、0x14(IOA,C,D)、0x18(TMR1)、0x1C(Predivider) 、0x20(TMR2) 、0x24(KI) 、0x28(RFC) 、0x2C(TMR3) 來符合 IC 中斷向量。

中斷的宣告如下：

```
void Interrupt INT_Intrerrupt(void) @ 0x10 {….}
void Interrupt IOA_Intrerrupt (void) @ 0x14 {….}
void Interrupt TMR1_Intrerrupt (void) @ 0x18 {….}
void Interrupt Predivider(void) @ 0x1C {….}
void Interrupt TMR2_Intrerrupt (void) @ 0x20 {….}
void Interrupt KI_Intrerrupt (void) @ 0x24 {….}
void Interrupt RFC_Intrerrupt (void) @ 0x28 {….}
void Interrupt TMR3_Intrerrupt (void) @ 0x2C {….}
```

Interrupt Service Routine 必須不含任何參數；否則編譯器將產生錯誤。

範例：

```
void interrupt T1_interrupt(void) @ 0x18
{
  int a=10;
  a += 20;
}

main()
{

  int i=0;

  asm("tmsx 0100000111b");
  /*
     * cs3 cs2 cs1 cs0   clock source of each timer
     *  0   0    0   1          ~PH3
     *  Set value = 000111b=0x07
     *  error = 1;
     *  set Time= (Set value +error)* 2^3 * 1/fosc (KHz)(ms)
   */

  asm("SF $80");     // Open Timer1 reload function
  asm("SIE* 0x02");  // Enable Timer1 Interrupt flag

  while(1)
  {
    ++i;
  }
}
```

在上述範例，當 TMR1 中斷相關設定已設定，然後用一個無限迴圈讓 TMR1 溢位。這會執行中斷服務。

注意：不同於普通函式的用法，當符合中斷條件，中斷應用將自動執行。

中斷被觸發並且在真正執行中斷服務程序之前，若在實際應用例子中需要將運算相關 Data RAM 資料預先儲存起來（尤其當該中斷服務程序之中，會變動到運算相關暫存資料內容時），並在執行完中斷服務程序後能回存運算資料。如此讓控制程序能在中斷後，仍然持續且正確的執行下去，意即運算結果不因執行中斷而受影響並且造成錯誤。

請注意，儲存（**ISR_SaveData**）及回存（**ISR_RestoreData**）組語程式已於 runtime_89.lib 中實作出來，使用者可依實際應用來呼叫上述二個程式。

以下範例為觸發中斷之 C 程序及所進入的中斷服務程序，其呼叫 ISR_SaveData()及 ISR_RestoreData()以儲存及回存運算相關 Data RAM 資料之程式片斷，供使用者參考：

C 程式 ：counter_Function() 為實際執行之中斷服務程序。

```
void  predivider_initial(void)
void  counter_Function(void);
```

```
// Function prototype of asm codes
void ISR_SaveData(void);
void ISR_RestoreData(void);

main()
{
        predivider_initial();
        while(1)
        {
          …..   // do something
        }
}

void interrupt counter_Interrupt(void) @ 0x1c
{
       asm("plc     0x08");
       ISR_SaveData();        // Save data
       counter_Function();
       ISR_RestoreData();   // Restore data
       asm("SIE*    0x0a");
}
// Initial process
void  predivider_initial(void)
{
       asm("plc 0x08");
       asm("sie*  0x9a");

}

void  counter_Function(void)
{
     ….. // do something
}
```

以下為實作在 runtime_89.lib 中之儲存及回存運算相關 Data RAM 資料(資料位址範圍為 0x2D0~0x2FF)之組語程式（請勿任意更動位址設定以避免錯誤！）：

```
.autoimport on
.importzp op1
.export _ISR_SaveData,_ISR_RestoreData

__RxAC__ .equ 0x4E
__RxCF__ .equ 0x4F

#define __INDEX_No__ 5

 ; op1 save to 0x2D0~0x2D7
 ; op2 save to 0x2D8~0x2DF
 ; op3 save to 0x2E0~0x2E7
 ; op4 save to 0x2E8~0x2EF
#define __opx_save_addr__ 0x2D0 ;0x2D0 ~ 0x2EF, total = 16 Bytes
```

```
#define __stkptr_save_addr__ 0x2F0 ; 0x2F0~0x2FF


.proc _ISR_SaveData

  sta __RxAC__
  ;;;   mmh 1h       ; //save MU to RAM address 01
  maf __RxCF__      ; save CF to RAM address 02
  rzr __INDEX_No__  ; save ZR to the 5th set backup RAMs
  rhl __INDEX_No__  ; save HL to the 5th set backup RAMs

  SHLX
  SETDAT op1
  SZRX
  SETDAT __opx_save_addr__
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH  @ZR,@HL
  SHLX
  SETDAT stkptr
  SZRX
  SETDAT __stkptr_save_addr__
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH  @ZR,@HL
  RTS

.endproc
.proc _ISR_RestoreData


  SHLX
  SETDAT __opx_save_addr__
  SZRX
  SETDAT op1
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
  LIDH$ @ZR,@HL
```

```
LIDH  @ZR,@HL
SHLX
SETDAT __stkptr_save_addr__
SZRX
SETDAT stkptr
LIDH$ @ZR,@HL
LIDH$ @ZR,@HL
LIDH$ @ZR,@HL
LIDH  @ZR,@HL

mhl __INDEX_No__  ; //restore HL
mzr __INDEX_No__ ; //restore ZR
mra __RxCF__     ; //restore CF
;; smui 1h       ; //restore MU
lda __RxAC__     ; //restore ACC
RTS

.endproc
```

注意：在中斷應用時，請留意以下三項限制。

1. 不允許巢狀之中斷程序呼叫（即在未離開某一中斷程序前又進入另一中斷程序去執行）。
2. 若中斷程序需要宣告變數來運算時，建議不要宣告為區域變數，而改宣告為全域變數。以節省堆疊（stack）空間。
3. 若控制中斷的程式正在進行浮點數運算時，建議此時不要被中斷而進入中斷程序。因為此動作將可能造成浮點數運算結果錯誤。

6. 前置處理器（Preprocessors）

前置處理器是編譯器所引用的程式，在編譯之前處理程式碼。為與其他程式碼本文做區分，前置處理程式的指令是原始檔案中每一行起始字元為 # 的句子。前置處理的程式原始碼，為中間檔案，因為這將輸入到編譯器，所以必須為有效的 C 語言程式。

前置處理器的指令和巨集延展的相關物件，將會在此章節裡面討論。在前置處理指令概論後，主題將包含 include 本文巨集，檔案包含，條件式編譯指令，和附註。

前置處理器是由以下指令所控制的：

指令	說明
#define	定義一個巨集
#undef	移除前置處理器的巨集定義
#include	插入從其他原始檔案的文字
#if	依照常數運算式的運算結果，條件式隱藏某部分的原始程式碼
#ifdef	若一個巨集名稱被定義，條件式決定包含原始文字
#ifndef	若一個巨集名稱未定義，條件式決定包含原始文字
#else	若之前的#if，#ifdef，#ifndef，或#elif測試失敗，條件式決定包含原始文字
#elif	#ifndef或 #elif測試失敗，根據常數運算式的結果值
#endif	終止條件式文字

巨集定義（Macro Definition）

前置處理器定義指令，讓前置處理器藉由指定的標記，來取代所有後續出現的巨集。

非參數的巨集定義（Non-parameter Macro Definition）

#define 指令可以包含類似物件的定義或類似函式的定義。

#define versus const_value

#define 指令可以用來創造一個數字、字元或固定字串的名稱，而任何類型的 const 物件亦可被宣告。

參數巨集的定義（Definition of Macro with Parameters）

一個巨集的參數可以是空的（包含 0 個前置處理的式子）。

例如，

```
#define SUM(a,b,c) a + b + c
SUM(1,2,3)                    // 1 用來取代 a，2 取代 b，且 3 取代 c */
```

包含檔（Files Include）

一個前置處理器的 include 指令，讓前置處理器以指定檔案的內容來取代指令。前置處理器 #include 指令有以下的格式：

```
    #include <file1.h>
或
    #include “file1.h”
```

例如：

```
    #include <july.h>
```

條件式編譯（Conditional Compile）

前置處理器條件式編譯指令，讓前置處理器條件性地決定編譯或不編譯某部份的原始程式碼。這些指令檢查一個常數運算式或識別符號，來判斷哪些式子是可通過前置處理器、哪些式子是前置處理過程中是要跳過的。這些指令如下：

- #if
- #ifdef
- #else
- #ifndef
- #elif
- #endif

7. 在 C 專案中混用 C、組語程式碼

基本概念

一般而言以 C 語言來開發單晶片的應用程式中，需要呼叫組語函式的情況有二種：

（1） 單晶片的一些特殊指令操作，無法藉由標準 C 語言的語法來描述。

（2） 為了實現單晶片系統所強調的即時控制性，必要時需引用組語指令以實現**部份**程式碼，來提高程序運行的效率。

如此，就會在一個 C 專案中同時出現 C 和組語混合編程的情況。在此章節中，我們將逐一討論混合編寫程式的基本方式與經驗分享，也請參考 "附錄" 章節中的例子以進一步了解實際應用。

在 TM89 C 編譯器中，C 程式與組語之間混用的機制，是以直覺的方式進行。 以下分三部份說明：

1. 在 C 程式中直接內嵌 inline 組語指令（不再次說明，請參考 " asm 宣告" 章節）。

2. 在 C 程式中呼叫組語函式

- 在 *.c 檔中的 C 程式碼，以 **函式原型（function prototype）**宣告在組語程式中所匯出之函式。例外情況為，組語函式無需參數傳遞時，可選擇性不在 C 程式碼中宣告組語函式原型。
- 在 *.asm 檔中的組語，使用 **export** 關鍵字匯出的函式類型有二:
  (1) 匯出標籤（**label**）。
  (2) 匯出以 **.proc / .endproc** 關鍵字所定義之組語函式 。

3. 在組語程式中呼叫 C 函式:

- 在 *.c 檔中的 C 程式碼中宣告及定義函式。
- 在 *.asm 檔中的組語以 call *_FunctionName* 呼叫 C 函式。

若組語函式有回傳值時，其回傳值將存在 @HL 索引暫存器。

當 C 與組語程式編輯完成後，需將相關之 C 語言、組語程式檔及相關程式庫檔案，加入到專案管理員的樹狀結構中，便於以專案為單位進行編譯。C 程式與組語之間相互呼叫函式的方式，各別又區分為有參數傳入或無參數傳入二種類型。在以下子章節中，將依序說明並在附錄中舉例說明。

C 程式呼叫無需傳入參數之組語函式

若沒有參數需要在 C 與組語程式呼叫間傳遞時，請參閱附錄例子 2。

C 程式呼叫需傳入參數之組語函式

當 C 程式呼叫組語函式時，需要傳遞參數時。 傳入組語函式的參數之設定順序是 "由右至左"，且組語函式中宣告的區域變數之定址順序是 "由下至上"。在組語中參數與變數的定址是相對性的。見以下簡單例子說明：

C 程式呼叫端 ：

```
int Foo(int, int*);
main()
{
        int a=10, b=20,c=0;
        c=Foo(a,&b);
}
```

組語函式：

```
.CPU TM8959

.AUTOIMPORT ON
.CASE ON

;.RAM
;.ENDRAM

;.RODATA

.importzp op1
.export _Foo
.declfunc   Foo(0,8)
.CODE
.proc _Foo
        SHLX
        SETDAT  Foo_LOCAL+4
        SZRX
        SETDAT  op1
        LID8$   @ZR,@HL
        LID8$   @ZR,@HL
        SHLX
        SETDAT  Foo_LOCAL+0
        SZRX
        SETDAT  $0080
        LID8$   @ZR,@HL
        LID8$   @ZR,@HL
        MHL     0
        SZRX
        SETDAT  op2
        LID8$   @ZR,@HL
        LID8$   @ZR,@HL
        CALL    runtime_intadd
        RTS
.endproc
.END
```

其中 .declfunc Foo(0,8)，第一個參數意指**區域變數**所佔之記憶體大小；第二個參數意指**傳入參數**大小（nibble 為單位）。上例中，依傳入參數之各個資料型態，來計算記憶體大小 8 = 2*2 (int) + 2*2(char*) 。（資料型態大小請參考" 宣告"章節）

區域變數在組語函式中命名方式

*FunctionName*_LOCAL

組語
Foo_LOCAL

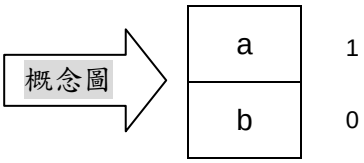
參數之設定順序是"由右至左"，所以其在組語函式中所對應的起啟位址如下表

*FunctionName*_PARAM

參數傳入方向　　參數定址順序

C語	組語
b	Foo _PARAM + 0
a	Foo _PARAM + 1

概念圖



請參閱附錄例子 3.

組語呼叫 C 函式

純組語檔案（*.asm）亦有可能呼叫 C 語言所宣告之函式，請參閱附錄例子 4。

C 和組語混合編程的一些經驗

C 專案中，C 和組語混合編程可以提高單晶片應用程序的運行效率，並使得軟體與硬體之間有最佳的配合。在此，分享一些由實際應用中所得到的一些經驗。

（一）謹慎使用組語指令

相對於組語，以 C 語言編程具以下優勢：提高開發效率、以自然語言的方式來編輯指令和語句、易於管理和維護的模組化程序以及程序具有在不同平台間的可攜性（portable）。因此，強烈建議在 C 語言程式中，儘量避免內嵌 inline asm 或全部以組語指令編寫模組程式。

TM89 C 編譯器對資料儲存空間的利用率，也肯定比使用者以手動設定變數或參數位址時的利用率要來得有效率，並且能減少重覆定址而造成隱藏而無法直接識別的錯誤。同時，C 語言提供了完善的函式庫，多樣與直覺性的控制和運算功能。因此，除了一些十分強調單晶片時效性的程式碼或 C 語言無法支援的操作，可以考慮以組語指令來實作外，其它部分仍建議應該以 C 語言來編寫。

（二）儘量以內嵌 inline asm 取代

這和上面所稱"謹慎使用組語指令"的說法並不矛盾。因為在實際應用中，若相對與 C 語言實作，而改以組語指令來實現部份程式碼，其確實可以提高運行效率。則當然儘量使用內嵌 inline asm 語句來實作。但我們依然強烈建議避免編寫"純組語檔案"（*.asm 檔）。

類似於純組語檔案的程式碼，其仍然可能在 C 語言架構下實現；方法為以 C 標準語法來定義所有的變數和函式名稱（包含需要傳遞的形式參數及區域變數）和最後需返回的參數語句，但函式內容的指令是用內嵌 inline asm 指令編寫。換言之，即以函式語法來包裝 inline asm。 如此，函式的運行效率和以純組語來編寫程式的運行效率幾乎是一模一樣的，所不同的是，各參數的傳遞格式是統一由 C 語言標準語法來實現，如此，可提高管理及維護的方便性。

8. 建立函式庫

函式庫

不同於執行檔，函式庫（library）不是單一獨立的程式碼，而是向其它程式提供服務的代碼。函式庫由數個可重定址（relocatable）的物件模組所組成，函式庫的副檔名為*.lib。使用者可將一系列相關運算的函式，集中、建立為一函式庫，便於讓其它程式呼叫，或是直接引入 TM89 C 所提供的函式庫來使用。如此，以達到程式碼再利用（reusable）的效益，減少重覆編碼的負擔。

使用函式庫

若實作的是簡單且小型的應用程式時，並不建議在程式中引用函式庫的內容，因為經由編譯與鏈結的過程中，會將函式庫內所有的函式內容整合到執行檔中。如此對小程式而言，反而需要更多的系統資源；在載入內部記憶體時也會消耗更多的時間。
在開發大型應用程式時，使用函式庫的機制將提供以下優點：

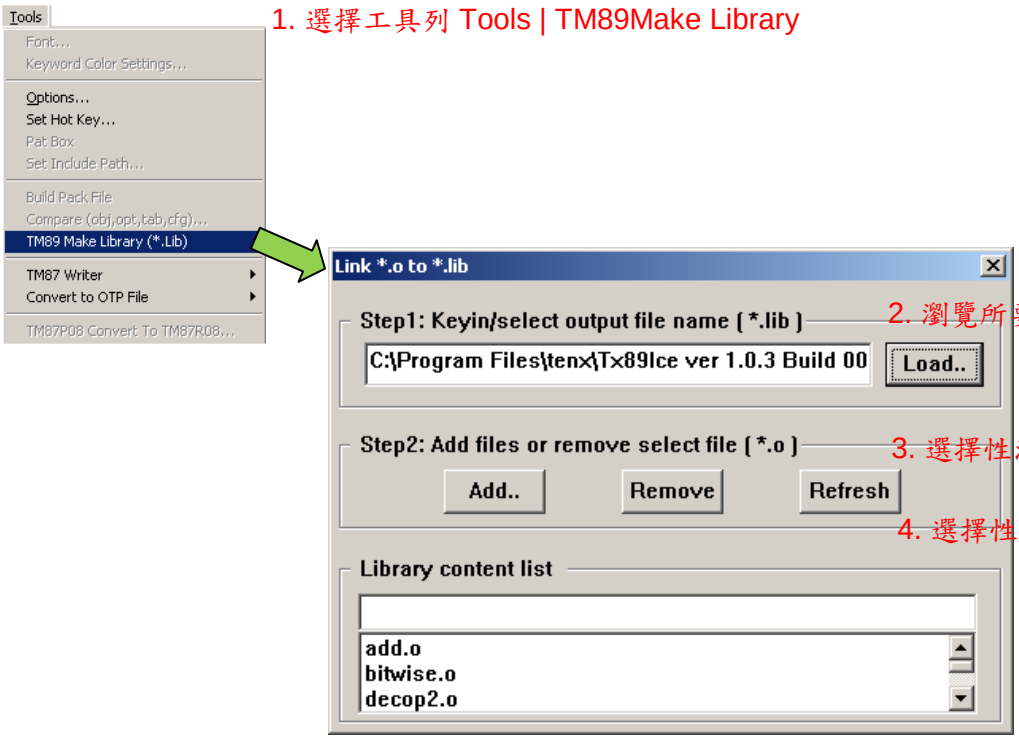
- 將相關運算模組函式集中在單一函式檔中，使得程式管理與維護更為簡單。
- 減少函式重複開發的時間，在函式文件說明上也更為有組織。
- 達到程式共享的目的，有效率的開發系統並縮短開發時程。

建立函式庫之方式

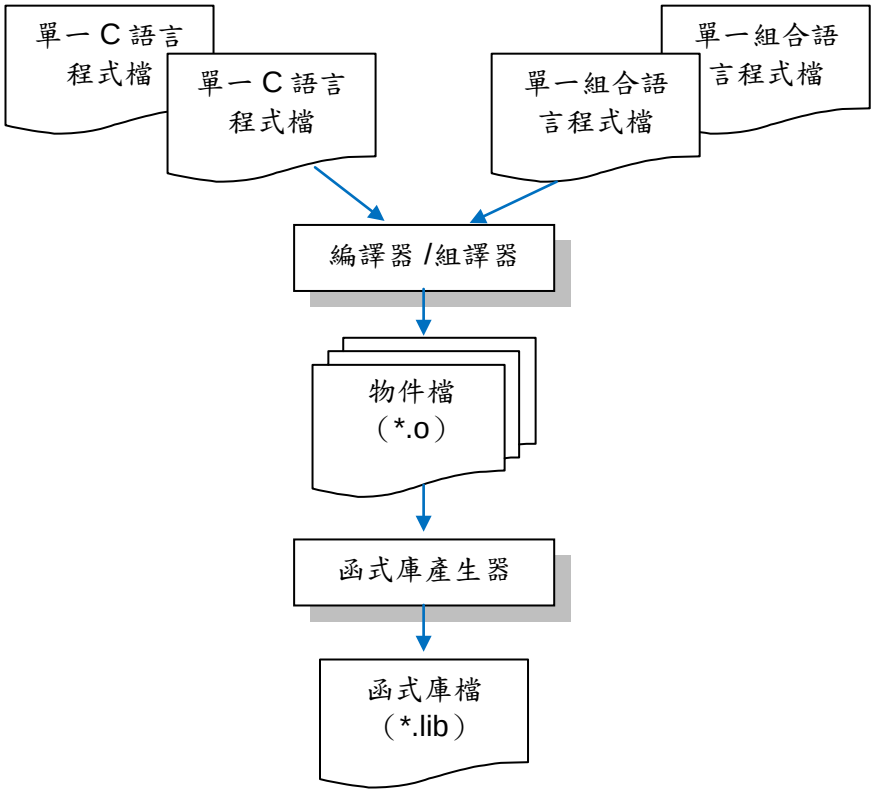
C 語言或組語函式皆可藉由 Tx89 IDE 所提供之工具以建立函式庫檔，其方法與概念如下圖：

方法分為二階段：

1. **產生物件檔**，將包含數個函式模組的單一 C 語言或組語程式檔，經過編譯器、組譯器處理，以產生物件檔（*.o）。
2. **建立函式庫檔**，利用 Tx89 IDE 所提供之工具：函式庫產生器（library maker），選擇性決定要將那些物件檔，集中建立為單一之函式庫檔（*.lib）。

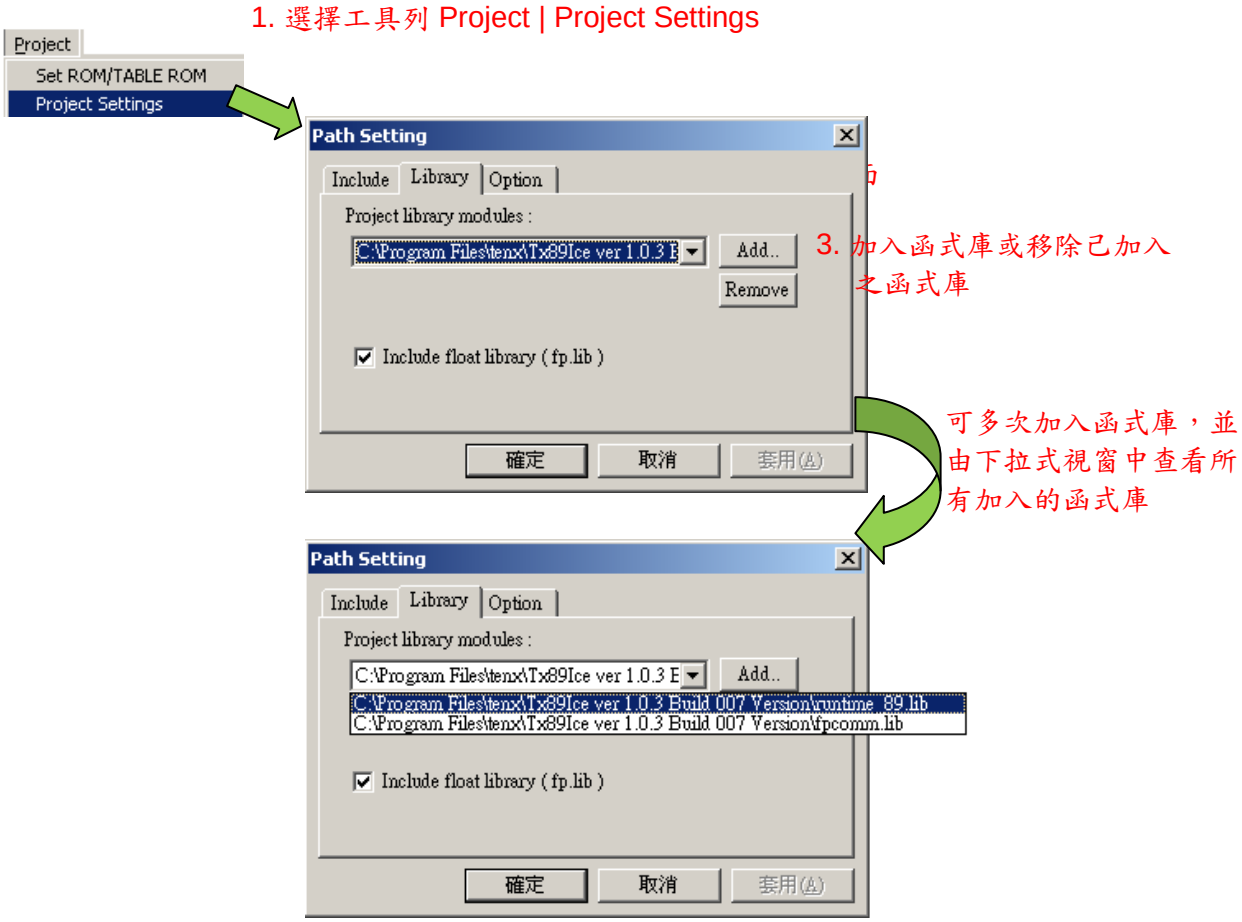

Tools
Font...
Keyword Color Settings...
Options...
Set Hot Key...
Pat Box
Set Include Path...
Build Pack File
Compare (obj,opt,tab,cfg)...
TM89 Make Library (*.Lib)
TM87 Writer
Convert to OTP File
TM87P08 Convert To TM87R08...
1. 選擇工具列 Tools | TM89Make Library
Link *.o to *.lib
Step1: Keyin/select output file name (*.lib)
C:\Program Files\tenx\Tx89Ice ver 1.0.3 Build 00
Load..
2. 瀏覽所要建立之函式庫檔名
Step2: Add files or remove select file (*.o)
Add..
Remove
Refresh
3. 選擇性決定要加入的物件檔 (可多選)
4. 選擇性決定要刪除的物件檔
Library content list
add.o
bitwise.o
decop2.o

概念圖：


單一C語言程式檔
單一C語言程式檔
單一組合語言程式檔
單一組合語言程式檔
編譯器/組譯器
物件檔 (*.o)
函式庫產生器
函式庫檔 (*.lib)

如何引用函式庫

在同一專案中的 C 語言或組語程式檔，其中會引用到某個函式庫中的某些函式模組時，請依循下列步驟來引入函式庫：



9. 附錄

例子 1

- 計算字串長度。

```
int strlen(char *dest) //包含結束字元null
{
        // 暫存dest位址的內容值
        asm("SZRX");
        asm("SETDAT %o", dest);
        asm("SHLX");
        asm("SETDAT $0080");
       asm("LID8$ @HL, @ZR");
        asm("LID8$  @HL, @ZR");
        asm("MHL 0");

        // 設定計算字元數的位址
        asm("SZRX");
        asm("SETDAT op2");
        asm("RZR 0");
        asm("LDS8# @ZR,$00");
        asm("LDS8# @ZR,$00");
        asm("MZR 0");

        // 計算字元數量
        asm("L1_calcute:");
        asm("LDA# @HL");
        asm("OR# @HL");
        asm("JZ          EXIT");
        asm("INC* @ZR");
        asm("JC Over_16");
        asm("JMP L1_calcute");

        // RETURN 字元數量
        asm("EXIT:");
        asm("SHLX");
        asm("SETDAT op2");
        asm("INC*# @HL");     //註解後，就不算結束字元
        asm("RTS");

        // 超過15個字元
        asm("Over_16:");
        asm("INC* op2+1");
        asm("JC Over_256");
        asm("JMP L1_calcute");

        // 超過255個字元
        asm("Over_256:");
        asm("INC* op2+2");
```

```
        asm("JC Over_4096");
        asm("JMP L1_calcute");

        // 超過4095個字元
        asm("Over_4096:");
        asm("INC* op2+3");
        asm("JMP L1_calcute");
}
```

- 複製來源字串（指標 src 所指之字串）到目的地字串（指標 dest 所指之字串）。

```
char* strcpy(char *dest,char *src) // copy src to dest
{
    asm("SZRX");
    asm("SETDAT  %o",src);// // Set offset of LOCAL name src
    asm("SHLX");                        // Read Value in src address
    asm("SETDAT $0080");
    asm("LID8$   @HL,@ZR");
    asm("LID8$   @HL,@ZR");
    asm("SZRX");
    asm("SETDAT  %o",dest); // Set offset of LOCAL name dest
    asm("SHLX");
    asm("SETDAT $00C0");
    asm("LID8$   @HL,@ZR");
    asm("LID8$   @HL,@ZR");
    asm("MHL 0");
    asm("MZR 0");
    asm("L1:"); // generate label name L1
    asm("LID8%%   @ZR,@HL");
    asm("LDA#  @HL");
    asm("OR#  @HL");
    asm("JZ EXIT");
    asm("JMP L1");
    asm("EXIT :"); // generate label name EXIT
    asm("SHLX");
    asm("SETDAT  %o",dest); //  return dest pointer in HL index.
    asm("RTS");
}
```

- 比較兩字串是否相同。

```
int strcmp(char *dest,char *src) // Compare dest and src
{
        asm("SZRX");
        asm("SETDAT %o", src);
        asm("SHLX");
        asm("SETDAT $0080");
        asm("LID8$ @HL, @ZR");
        asm("LID8$ @HL, @ZR");
```

```
asm("SZRX");                    // 讀取dest,src內容位址內的值
asm("SETDAT %o", dest);
asm("SHLX");
asm("SETDAT $00C0");
asm("LID8$ @HL, @ZR");
asm("LID8$ @HL, @ZR");

asm("L1:");
asm("MHL 0");
asm("MZR 0");
asm("MWR# op3,@ZR");
asm("MWR# op3+1,@ZR");

// 判斷字元Low的部份
asm("LDA# @HL");
asm("SUB  op3");
asm("JC $+2");
asm("DEC* op3+1");
asm("MAF op1");

// 判斷字元High的部份
asm("LDA# @HL");
asm("SUB op3+1");
asm("MAF op2");
asm("AND* op1");
asm("RHL 0");
asm("RZR 0");
asm("JB3 $+2");                     //判斷CF是否為1，是的話跳到PC+2
asm("JMP Ret_MinusOne");            //CF為0時，跳到 Ret_MinusOne
asm("JB2 LOOP");                    //判斷ZF是否為1，是的話跳到LOOP
asm("JMP Ret_One");         //ZF為0時，跳到Ret_One

asm("LOOP:");
asm("LDA# @HL");            //判斷src是否到底
asm("OR# @HL");
 asm("JNZ L1");

asm("LDA# @ZR");            //判斷dest是否到底
asm("OR# @ZR");
asm("JZ Ret_Zero");
asm("JMP L1");

asm("Ret_Zero:");
asm("SHLX");
asm("SETDAT op2");
asm("RHL 0");
asm("LDSH @HL");
asm("SETDAT $0000");
asm("MHL 0");
asm("RTS");
```

```
        asm("Ret_MinusOne:");
        asm("SHLX");
        asm("SETDAT op2");
        asm("RHL 0");
        asm("LDSH @HL");
        asm("SETDAT $FFFF");
        asm("MHL 0");
        asm("RTS");

        asm("Ret_One:");
        asm("SHLX");
        asm("SETDAT op2");
        asm("RHL 0");
        asm("LDSH @HL");
        asm("SETDAT $0001");
        asm("MHL 0");
        asm("RTS");
}
```

- 將 src 字串連接在 dest 字串之後，但只取 src 字串的前 n 個字元。

```
char* strncat(char *dest,char *src, int n)
{
        // SAVE n to op1
        asm("SZRX");
        asm("SETDAT %o", n);
        asm("SHLX");
        asm("SETDAT op1");
        asm("LID8$ @HL, @ZR");
       asm("LID8$ @HL, @ZR");

        // 暫存dest內容值
        asm("SZRX");
        asm("SETDAT %o", dest);
        asm("SHLX");
        asm("SETDAT $00C0");
        asm("LID8$ @HL, @ZR");
       asm("LID8$ @HL, @ZR");
        asm("MZR 0");

        //L1_Dest_End //尋找dest字串結尾的位址
        asm("L1_Dest_End:");
        asm("LDA @ZR");
        asm("RZR 0");
        asm("IDC%%");
        asm("OR @ZR");
       asm("JZ L2_Strcat");
        asm("JMP L1_Dest_End");

        //L2_Strcat // 暫存src內容值
```

```
        asm("L2_Strcat:");
        asm("SZRX");
        asm("SETDAT %o", src);
        asm("SHLX");
        asm("SETDAT $0080");
        asm("LID8$ @HL, @ZR");
       asm("LID8$ @HL, @ZR");
        asm("MHL 0");
        asm("MZR 0");

        // L3_Src_End //Copy n char of src to dest
        asm("L3_Src_End:");
        asm("LID8%% @ZR,@HL");  // Copy Src string to Dest String
        asm("LDA# @HL");
        asm("OR# @HL");
        asm("JZ EXIT");
        asm("DEC* op1");
        asm("LDA op1");
        asm("OR op1+1");
        asm("OR op1+2");
        asm("OR op1+3");
        asm("JZ EXIT");
        asm("JMP L3_Src_End");

        // generate label name EXIT
        asm("EXIT:");
        // 判斷最後是否為結束字元
        asm("RZR 0");
        asm("LDA# @ZR");
        asm("OR @ZR");
        asm("JNZ Add_null");

        asm("Ret:");
         asm("SHLX");
         asm("SETDAT  %o",dest); //  return dest pointer in HL index.
         asm("RTS");

       asm("Add_null:");
       asm("MZR 0");
       asm("LDS8 @ZR,$00");
       asm("JMP Ret");
}
```

- 將原始字串的內容（指標 src 所指之字串）串接在目的地字串（指標 dest 所指之字串）之後。

```
char* strcat(char *dest,char *src)
{
        // 暫存src內容值
        asm("SZRX");
        asm("SETDAT %o", dest);
```

```
            asm("SHLX");
            asm("SETDAT $00C0");
            asm("LID8$ @HL, @ZR");
           asm("LID8$ @HL, @ZR");
            asm("MZR 0");

           //label : L1_Dest_End
           //尋找dest字串結尾的位址
            asm("L1_Dest_End:");
            asm("LDA @ZR");
            asm("RZR 0");
            asm("IDC%%");
            asm("OR @ZR");
           asm("JZ L2_Strcat");
            asm("JMP L1_Dest_End");

             //label : L2_Strcat
             // 暫存src內容值
            asm("L2_Strcat:");
            asm("SZRX");
            asm("SETDAT %o", src);
            asm("SHLX");
            asm("SETDAT $0080");
            asm("LID8$ @HL, @ZR");
           asm("LID8$ @HL, @ZR");
            asm("MHL 0");
            asm("MZR 0");

            //label : L3_Src_End
           //判斷src字串是否到達結束字元
            asm("L3_Src_End:");
            asm("LID8%% @ZR,@HL");
            asm("LDA# @HL");
            asm("OR# @HL");
            asm("JZ EXIT");
            asm("JMP L3_Src_End");

            asm("EXIT:");       // generate label name EXIT
           asm("SHLX");
           asm("SETDAT  %o",dest); //  return dest pointer in HL index.
           asm("RTS");
}
```

● 找出原始字串（指標 src 所指之字串）在目的地字串（指標 dest 所指之字串）的位置。

```
char* strchr(char *dest,char src) // Find Src in dest  string
{
        // 暫存dest內容值
        asm("SZRX");
        asm("SETDAT %o", dest);
```

```
        asm("SHLX");
        asm("SETDAT $0080");
        asm("LID8$ @HL, @ZR");
        asm("LID8$ @HL, @ZR");
        asm("MHL 0");

        asm("SZRX");                    //Read charator src
        asm("SETDAT %o", src);
        asm("RZR 0");

        asm("Chr_Compare:");   //Compare *dest and src low
        asm("MZR 0");
        asm("MHL 0");
        asm("LDA @ZR");
        asm("EOR @HL");
        asm("JZ Compare_Next");
        asm("IDC8&");
        asm("JMP LOOP");

        asm("Compare_Next:"); // Compare *dest and src high
        asm("IDC$");
        asm("LDA @ZR");
        asm("EOR @HL");
        asm("JZ EXIT");
        asm("IDC&");
        asm("JMP LOOP");

        asm("LOOP:");                   //判斷dest是否到底
        asm("RHL 0");
        asm("LDA# @HL");
        asm("OR @HL");
        asm("JZ Ret_ZERO");
        asm("JMP Chr_Compare");

        asm("EXIT:");                   // Return Match address
        asm("SHLX");
        asm("SETDAT $0080");
        asm("RTS");

        asm("Ret_ZERO:");               // Return Null
        asm("SHLX");
        asm("SETDAT op2");
        asm("LDSH @HL");
        asm("SETDAT $0000");
        asm("RTS");
}
```

例子 2

C 程式呼叫沒有參數需要傳遞與回傳值的組語函式 Loop_10_Times。

C 語言：直接呼叫組語函式

因為呼叫過程中無需傳入參數及回傳值，故可選擇性決定是否要宣告組語函式原型，並且直接呼叫函式即可。

```
void Loop_10_Times(void);

main()
{
  int i=0,a=0;

  for(;i<5;i++)
  {
          ++a;
          Loop_10_Times();
  }
}
```

組語函式：

```
.CPU TM8959

.AUTOIMPORT ON
.CASE ON

.RAM
.ENDRAM

.RODATA

.importzp op1
.export _Loop_10_Times
.declfunc   Loop_10_Times(0,0)
.CODE
.proc _Loop_10_Times
          SZRX
          SETDAT  op1
          RZR     0
          LDS8#  @ZR,$0A
          LDS8#  @ZR,$00
          MZR     0

LOOP:
          DEC*    @ZR
          JNZ     LOOP
          RTS

.endproc
```

```
.END
```

例子 3

C 語言：宣告函式原型，並呼叫組語程式所匯出之 strcpy()

C 程式呼叫一由組語所宣告及定義之 strcpy 函式，該函式需傳入二個字元指標以進行字串複製，但沒有回傳值。

```
void strcpy(char*,char*);          // Function Prototype for asm function
main()
{
  char String1[12] = "I like TM57";
  char String2[12] = "I like TM89";
  strcpy(String1, String2);        // call asm function
}
```

組語 strcpy() 函式

在函式定義之前，.export _strcpy 以關鍵字 **export** 匯出函式 _strcpy，並以關鍵字 **declfunc** 宣告函式 strcpy 的區域變數與傳入參數佔記憶體的大小（nibble 為單位），宣告語句為 .declfunc strcpy(0,8)。

```
  ;  string copy function by asm code
  ;*****************************************
  ; char* strcpy (char* dest,  char* src)
  ;*****************************************
        .autoimport     on
        .debuginfo      on
        .export         _strcpy ; leader char is  _ ( at prefix )
        ; use ".declfunc" directive to allocate the size of local and parameter.
        ; format: .declfunc funname(local_size,param_size)
        ; parameter count= source(0)+target(8) = 8 (nibbles)
        ; declared 8 nibbles space to parameter
        .declfunc       strcpy(0,8)

  .CODE

  .proc _strcpy ;*** parameter address stack counter is from right to left

        SZRX
        SETDAT strcpy_PARAM+0       ;Set offset of LOCAL name src
        SHLX
        SETDAT $0080
        LID8$   @HL,@ZR
        LID8$   @HL,@ZR
        SZRX
        SETDAT  strcpy_PARAM+4      ;Set offset of LOCAL name dest
        SHLX
        SETDAT $00C0
        LID8$   @HL,@ZR
```

```
        LID8$   @HL,@ZR
        MHL 0
        MZR 0
        L1:             ; generate label name L1
        LID8%   @ZR,@HL
        LDA#    @HL
        OR#     @HL
        JZ EXIT
        JMP L1
        EXIT            ; generate label name EXIT
        SHLX
        SETDAT  strcpy_PARAM+4   ; return dest pointer in HL index.
        RTS

.endproc
```

例子 4

在這例子中，組語程式呼叫 C 語言之 strcpy() 函式，以進行字串之複製。

C 語言：定義 strcpy()函式以供組語程式呼叫

```
void strcpy(char* des,char* source)
{
    int i=0;
    for (i=0;source[i] != '\0'; ++i)
    {
       des[i] = source[i];
    }
    return;
}
```

組語程式：呼叫 C 語言 strcpy()函式

因為 C 函式名稱在經過 C 編譯器編譯過後，原名稱將變成以 '_' 為前導的函數名稱（即，_strcpy）。故而就本例子而言，組語程式要呼叫 C 函式 strcpy 的語句需改為 call _strcpy。

在下列程式中，設定來源字串變數為 src_str，目的地字串變數為 tar_str，並將此二個變數位址分別對應到 strcpy_PARAM+0 及 strcpy_PARAM+4 之位址，以傳入 C 函式 strcpy 中做運算。使用者可讀取變數 tar_str 以查看運算結果（即，tar_str 結果值為 “ABCD”）。

```
;************************************* **
;*** call strcpy function from asm code
;************************************* **

 .autoimport on

 src_str = 300h
 tar_str = 30ah
```

```
SHLX
SETDAT  src_str
LDS8# @HL, ‘A’
LDS8# @HL, ‘B’
LDS8# @HL, ‘C’
LDS8# @HL, ‘D’
LDS8   @HL, 0

 ;*****************************************
 ;*** pass parameter from right to left
 ;*** one pointer size is 4 nibbles
 ;*****************************************
SHLX
SETDAT  strcpy_PARAM+0
LDS8#  @HL,src_str&0xFF
LDS8#  @HL,(src_str>>8)&0xFF
LDS8#  @HL,tar_str&0xFF
LDS8    @HL,(tar_str>>8)&0xFF
call  _strcpy

loop:
  nop
  jmp loop
```