TM57 C 編譯器

8-位元 微型控制器

使用手冊

tenx reserves the right to change or discontinue the manual and online documentation to this product herein to improve reliability, function or design without further notice. **tenx** does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. tenx products are not designed, intended, or authorized for use in life support appliances, devices, or systems. If Buyer purchases or uses **tenx** products for any such unintended or unauthorized application, Buyer shall indemnify and hold **tenx** and its officers, employees, subsidiaries, affiliates and distributors harmless against all claims, cost, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use even if such claim alleges that tenx was negligent regarding the design or manufacture of the part.

tenx technology inc.

Rev 1.3, 2022/06/30

AMENDMENT HISTORY

Version	Date	Description
V1.0	Oct, 2011	新頒
V1.1	Jul, 2012	1. 增加中斷保護:儲存及回存資料 2. 增加記憶體對應圖之章節 3. 增加位元變數定址在 bank1 之設定 4. 修正運算暫存器之定址敍述 5. 增加結構/聯合可指定位址宣告 6. 修正位元變數只支援全域變數宣告
V1.2	Sep, 2021	1. asm 宣告新增%n 格式及說明 2. 結構和聯合宣告新增 High Bytes/Low Bytes 提示。
V1.3	Jun, 2022	1. Pragma 新增 tableromdt 指令及說明(0.5.7 版本後支援)

目錄

AMENDMENT HISTORY **2**

1. TM57 系列 C 語言編譯器概述 **6**

關於 TM57 系列 C 語言編譯器 6
編譯 C 語言程式 7
詞彙約定 9
原始程式字元集 9
註譯 10
識別符號 10
關鍵字 11
常數 11
數字常數 11
字元常數 12
列舉常數 12
全域常數 12
字串常數 13
運算符號 13
標點符號 13

識別符號的意義 **14**

消除歧異的名稱 14
作用域（Scope） 14
區塊作用域（Block Scope） 14
函式作用域（Function Scope） 14
函式原型作用域 14
檔案作用域（全域作用域） 15
命名空間中的識別符號 15
識別符號的鏈結 15
儲存空間持續期間（Storage Duration） 15

2. 宣告（Declaration） **16**

儲存類別說明符 16
型態說明符（Type Specifiers） 18
Fplane / Rplane 宣告 18
結構和聯合宣告 21
結構、聯合中宣告和使用位元欄位 22
位元資料型態 23
位元運算元 25
列舉宣告 25
型態限定詞（Type Qualifiers） 26

宣告（Declarators） 26
指標宣告（Pointer Declarators） 27
矩陣宣告（Array Declarators） 28
函式宣告和原型 29
asm 宣告 30
宣告的限制 33
型別定義（typedef） 34
初始化（Initialization） 34
聚集的初始化（Initialization of Aggregates） 35
3. 算式和運算符號（Expressions and Operators） 36
在 C 語言的運算符號優先序和結合律規則 36
主要的式子（Primary Expressions） 37
後置式（Postfix Expressions） 37
矩陣索引符號運運算元（Array Subscripting Operator） 37
結構和聯合的參照（Structure and Union References） 38
間接結構和聯合參照（Indirect Structure and Union References） 38
後置++和後置--（Postfix ++ and Postfix --） 38
單元運運算元（Unary Operators） 39
位址或間接引用運運算元（Address-of and Indirection Operators） 39
單元運運算元+和-（Unary + and Unary – Operators） 39
邏輯否定！和位元否定 ~ 運運算元 39
前置 ++ 和 -- 運運算元 40
sizeof 單元運運算元 40
乘法運運算元（Multiplicative Operators） 40
加法運運算元（Additive Operators） 41
位移運運算元（Shift Operators） 41
關係運運算元（< > <= >=） 42
相等運運算元（== !=） 42
邏輯運運算元 AND（&&），邏輯運運算元 OR（||） 42
條件運運算元（Conditional Operator） 43
4. 述句（Statements） 44
算式述句（Expression Statements） 44
區塊述句（Block Statement） 44
選擇述句（Selection Statements） 44
if 述句 44
switch 述句 45
重複述句 45
while 述句 46
do 述句 46
for 述句 47
jump 述句 48

goto 述句 48
continue 述句 48
break 述句 49
return 述句 49
標籤述句 49
中斷 49
(一) R-Plane 51
(二) F-Plane Bank 0 52
(三) F-Plane Bank 1 52
ISR_SaveData、ISR_RestoreData 53
ISR_SaveData_5、ISR_RestoreData_5 61

5. 前置處理器（Preprocessors） 64

巨集定義（Macro Definition） 64
非參數的巨集定義（Non-parameter Macro Definition） 64
參數巨集的定義（Definition of Macro with Parameters） 64
包含檔（Files Include） 65
條件式編譯（Conditional Compile） 65
pragma 指令 (#pragma) 65

在 C 專案中混用 C、組語程式碼 68

基本概念 68
C 程式呼叫無需傳入參數之組語函式 68
C 程式呼叫需傳入參數之組語函式 69
組語呼叫 C 函式 70
C 和組語混合編程的一些經驗 70
(一) 謹慎使用組語指令 70
(二) 儘量以內嵌 inline asm 取代 71
(三) 避免在 C/ASM 混合開發時, 使用.org xx 指令 71

6. 建立函式庫 72

函式庫 72
使用函式庫 72
建立函式庫之方式 72
如何引用函式庫 75

7. 記憶體對應圖 76

8. 附錄 77

例子 1 77
例子 2 81
例子 3 81
例子 4 83

1. TM57 系列 C 語言編譯器概述

關於 TM57 系列 C 語言編譯器

TM57 系列 C 語言編譯器符合 ANSI C 標準，可是 TM57 系列 C 語言編譯器不支援函式指標。此外，為使 tenx 晶片的工作效率與控制性能達到最佳狀態，以及提供給 C 語言程式設計者更好的程式設計支援，增加了以下特殊功能：

1. 位元變數

- 程式中只允許在全域範圍中宣告位元資料型態，其宣告語法請參考位元資料型態。
- 在結構、聯合中使用位元欄位，宣告語法請參考結構、聯合中宣告和使用位元欄位。

2. 為讓 C 語言程式設計者更自主性地安排全域變數（global variable）及函式地址，以切實地符合實作需求；TM57 C 語言編譯器提供指定全域變數定址在那一個暫存器位置（Fplane 或 Rplane），和指定全域常變數及函式在 TABLE ROM 地址的功能。若實作的晶片中，其 F-Plane RAM 有一個以上的 bank 時，亦可指定全域變數是要儲存在 F-Plane 的那個 bank 位置上。

- 宣告變數指定儲存暫存器之語法及應注意事項，請參考 F-Plane / R-Plane 宣告。
- 為全域常變數安排在 TABLE ROM 起始位址, 用戶可指定 #pragma tableromaddr 來達到目的。
- 在函式定義程式中指定 TABLE ROM 位址, 請參考函式宣告。

3. 提供中斷函式與多樣的中斷保護功能：觸發中斷函式的執行過程中，可能會變動到運作暫存器的內容而影響執行結果。tenx 除了提供具有自動儲存功能的晶片外，也提供多樣的中斷保護功能，讓使用者依不同的程式運算複雜度，有效率的決定儲存運作暫存器的內容。

- 程式運算複雜度與相關影響到的運作暫存器，其對應關係請參考運算暫存器。
- 運作暫存器之記憶體對應圖，請參考記憶體對應圖。
- 具備中斷保護功能之組語副程式，請參考中斷保護。
- 啟動中斷保護功能時，應注意事項請參考中斷注意事項。

4. 為了切合單片機的特殊指令操作特性與即時控制性，組合語言程式相對於 C 語言程式更能吻合單片機需求。因此，C 專案中允許 C 語言和組合語言混合程式設計的情況。

- 在 C 程式中直接內嵌組合語言指令(asm,__asm__)，宣告語法請參考 Asm 定義。
- 為避免變數、參數或函式名稱在組合語言指令中，可能引發的拼字與維護問題，建議使用格式符號來替換變數名稱，請參考格式符號。

- C 和組合語言程式混合程式設計的基本方式與實例說明：（1）在 C 程式中調用彙編函數，其中分為有/無傳參數來分別說明，（2）在組合語言程式中調用 C 函數。請參考在 C 項目中混用 C、組合語言程式代碼。
- 在何種情況下，最適合 C 和彙編混合程式設計方式的經驗分享，請參考 C 和彙編混合程式設計的一些經驗。

5. 支援函式庫

- C 語言或彙編函數皆可藉由 TICE99 IDE 所提供之工具以建立庫函數檔，逐步的建立方法請參考建立函式庫之方式。
- 引用庫函數的方式，請參考如何引用函數庫。

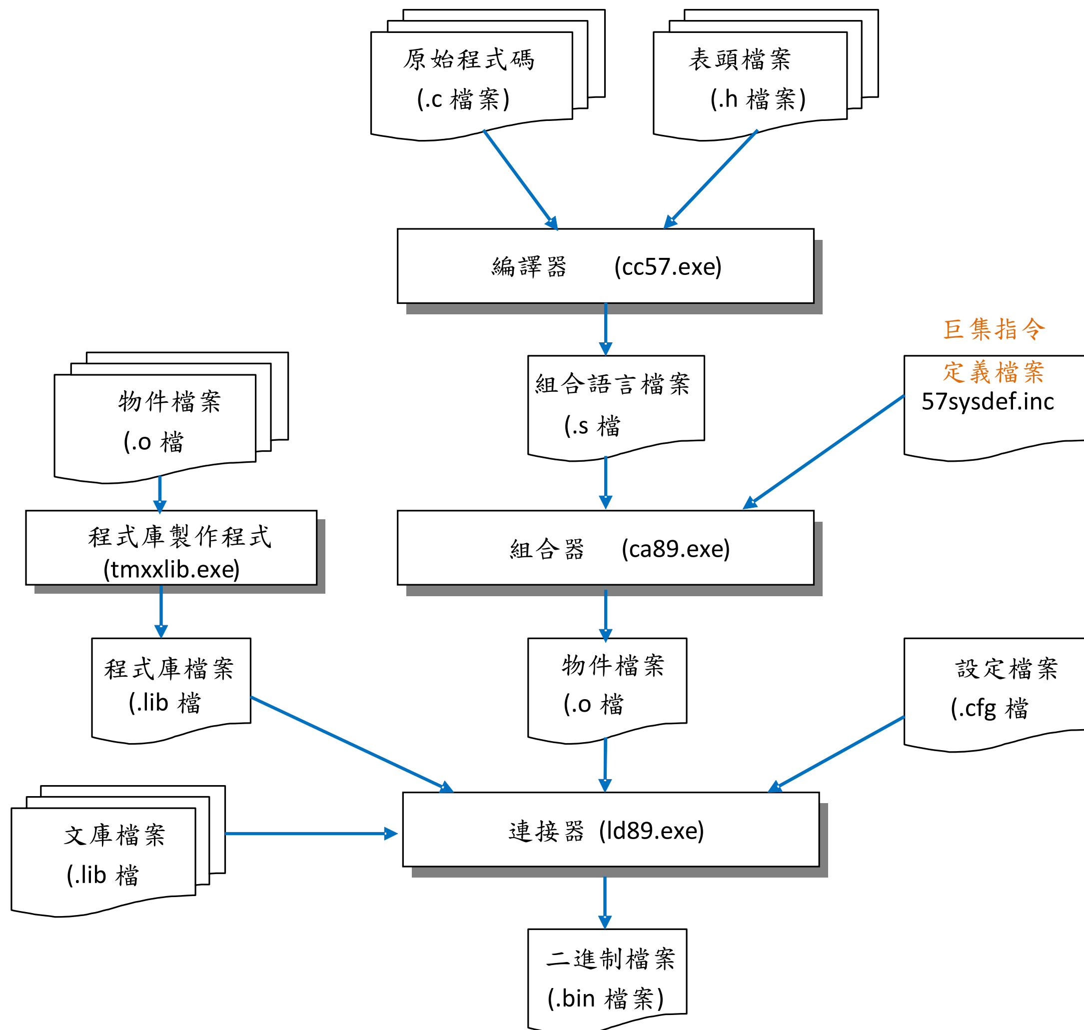
編譯 C 語言程式

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：

- **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。
- **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。
- **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”。
- **晶片相關檔案**：包含執行中使用的程式庫檔（runtime57_XXX.lib 檔案）和設定檔案（.cfg files）。這些檔案包含記憶體重新分配和每一類型晶片的指令集資訊。
- **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 (cc57.exe)
巨集指令
定義檔案
57sysdef.inc
物件檔案
(.o 檔
組合語言檔案
(.s 檔
程式庫製作程式
(tmxxlib.exe)
組合器 (ca89.exe)
程式庫檔案
(.lib 檔
物件檔案
(.o 檔
設定檔案
(.cfg 檔
文庫檔案
(.lib 檔
連接器 (ld89.exe)
二進制檔案
(.bin 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）和常數字元（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、移字元號、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	水準製錶
\v	垂直製錶
\’	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。您也可以用在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 TM57 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	for	static	**__asm__**
char	goto	struct	**interrupt**
continue	if	switch	**rplane**
default	int	typedef	**bit**
do	long	union	**FPLANE**
else	return	unsigned	**RPLANE**
			bank1

注意：

- **__asm__, interrupt, rplane, RPLANE, FPLANE, bank1 和 bit 為額外 MCU 的關鍵字**
- ***不支援 float 和 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，但為多餘的，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 TM57 系列 C 語言編譯器，全域常數變數儲存在 Program 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 0009~000a，且數據是和數據長度最後兩個位元組合起來的。以此為例，變數數據為 0x000A = 10。

```
const int gInt=10;
main()
{
  int kk;
  kk=gInt;
}
```

用戶也可使用 preprocessor directive: #pragma tableromaddr 來指定全域變數的起始位址。 例如，全域變數 array[6]想定址在 0x0100，而全域變數 gVar1 無需指定位址並交由 C-編譯器定址。c 程式如下圖所示。

```
3 #pragma tableromaddr (0x0100)
4 const unsigned char array[6] = {2,3,4,5,6,7};
5 #pragma tableromaddr (off)
6 const gVar1 = 23;
```

Var name	Dec Value	Addr/PC Range	Data Type	Bank no.
D:\bcb_test\fla80_cost...				
Global Var				
D:\bcb_test\fla8...				
array		PC: 0101~0106	const BYTE[6]	
[0]	2 ' '	PC: 0101~0101	const DATA_TYPE_BYTE	
[1]	3 ' '	PC: 0102~0102	const DATA_TYPE_BYTE	
[2]	4 ' '	PC: 0103~0103	const DATA_TYPE_BYTE	
[3]	5 ' '	PC: 0104~0104	const DATA_TYPE_BYTE	
[4]	6 ' '	PC: 0105~0105	const DATA_TYPE_BYTE	
[5]	7 ' '	PC: 0106~0106	const DATA_TYPE_BYTE	
gVar1	23	PC: 0008~0009	const int	

注意：請注意全域變數位址的安排，以避免產生位址衝突的錯誤.

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。TM57 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。

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，在此區域識別符號可被看見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（***label***）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幹擾，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。

2. 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式。

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鍵結和儲存持續期間。儲存類別說明符的表示方式如下：

storage-class-specifier:	static extern typedef rplane

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使化為 0。

rplane 宣告可以讓使用者使用指定在 rplane RAM 的全域變數（*global variable*）。

範例：

```
rplane int _gx;     // R plane變數_gx擺放在全域區域

 void main(void)
{
    int i,j;
    _gx = i+j;    // 存 i+j 到 R-plane RAM _gx
 }
```

注意：雖然有一些 TM57 系列晶片有提供 R-Plane RAM 寫入模式，可是不允許讀取 R-Plane RAM 數據。詳細請參考目前的 TM57 系列晶片之相關規格文件。

```
#define _PWRDOWN 0x03
unsigned char _powerdown @_PWRDOWN:RPLANE ;
rplane int _gx;
 void main(void)
{
      unsigned char uc;
      rplane int ri;          // 錯誤，必須放在全域區域

      uc=_gx;                // 編譯過程中一定會出現錯誤
                              // 若R-Plane RAM只是可寫模式（例如：TM57PA40）
      uc=_powerdown;    // 編譯/組譯過程中一定會出現錯誤
                                 // 若R-Plane RAM只是可寫模式（例如：TM57PA40）
}
```

型態說明符（Type Specifiers）

型態說明符如下列，其語法如下：

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

注意：不支援資料型態 *float* 和 *double*

Fplane / Rplane 宣告

在 TM57 系列，暫存器分為兩個記憶體區塊：F-Plane 和 R-Plane。這提供使用者一更方便的方式，將全域變數定址在指定的 R-Plane 或 F-Plane RAM。RAM 儲存宣告表示方式如下：

declaration: *type-specifier Var_name @Address* [*@bit*]:(**RPLANE**|**FPLANE**)

在"@"之後的 RAM 位址，可用十進制或十六進制表示。

範例：

<table>
<tr><th>C 程式碼</th><th>ASM 程式碼</th></tr>
<tr><td>

```
/* Specify the address of var: F_Addr to 0x30 in F-plane*/

// declare global variable
unsigned char F_Addr@0x30:FPLANE;

main()
{
    F_Addr=0x20;   /* initial value = 0x20 */
}
```

</td><td>

```
000000: 3001  GOTO     0x1
000001: 2005  CALL     0x5
000002: 1920  MOVLW   0x20
000003: 00B0  MOVWF   0x30
000004: 0040  RET
000005: 0040  RET
```

</td></tr>
</table>

TM57 C 語言編譯器提供 R-Plane 或 F-Plane 的位元運算，以維護 IC 中位元型態的暫存器。至於那些 IC 的記憶體位址範圍是可以被位元定址的，詳細請參考 IC 規格文件。位元運算範例如下：

```
/* Specify the address of var: Addr  to 0x30  in R-Plane*/

// declare global variable to access the bit field 0 of address 0x30 in R-Plane
unsigned bit Addr@0x30@0:RPLANE;

main()
{
    Addr = 0;    /* initial value = 0 */
}
```

注意：

1. 在 R-Plane，MOVWR 和 MOVRW 指令可用來做記憶體存取。因為以上指令資料單位元為 BYTE，因此我們不建議使用位元元方式的處理。
2. F-Plane 和 R-Plane 宣告，只允許使用者對全域變數定址。

在鏈結程式中，鏈結器將儲存運作暫存器到 F-Plane RAM 從 0x20 to 0x37（24 bytes），但實際暫存器的使用取決於操作的複雜度與程式中是否有宣告位元變數而不同。暫存器使用的最高範圍將分別依 MCU 類別做說明：

F-Plane RAM Bit Addressable = 8 bytes

程式中**宣告**位元變數：

位址	大小 (bytes)	操作暫存器	位址範圍
0x20~0x27	8	op1 ~ op2	固定
-	8	op3 ~ op4	非固定
-	1	tmp1	非固定
-	4	stkptr	非固定

程式中**無宣告**位元變數：

位址	大小 (bytes)	操作暫存器	位址範圍
0x20~0x2F	16	op1 ~ op4	固定
-	1	tmp1	非固定
-	4	stkptr	非固定

F-Plane RAM Bit Addressable >= 16 bytes

位址	大小 (bytes)	操作暫存器	位址範圍
0x20~0x2F	16	op1 ~ op4	固定
-	1	tmp1	非固定
-	4	stkptr	非固定

註：

1. 上文中所指之位元變數宣告意指**一般位元變數之宣告**（**非**指定 bank1 及 F-Plane、R-Plane 位址之位元變數宣告），意指需**由鏈結器決定位址之位元變數**，例如 bit bVal;。
2. 當程式中有宣告一般位元變數時，鏈結器定址 op1~op4，tmp1，stkptr 及位元變數的順序，以下將分二類來說明其分配位址之順序：

F-Plane RAM Bit Addressable = 8 bytes 或單一Bank	F-Plane RAM Bit Addressable >= 16 bytes
● 運算用到 op1~op3 或 op1~0p4 時 op1~op2 | 位元變數 | op3~op4 | tmp1 | stkptr ● 運算用到op1~op2 或op1時 op1~op2 | 位元變數 | tmp1 | stkptr	● 運算用到 op1~op# 時，#可能為2、3、4 op1~op# | 位元變數 | tmp1 | stkptr
● 運算無op1~op4時 位元變數 | tmp1 | stkptr	

以上位址範圍是預留為執行時期程式庫的運算，而且此位址用來儲存暫存結果。建議使用者請勿使用這些位址以防止意外的錯誤發生，**尤其是當一個專案裡面，同時使用 C 語言程式碼和 ASM 程式碼時**。強烈建議使用者使用一簡單的運算式子以降低運算複雜度。以下表格為各種算術運算可能會使用到的暫存器。

● 一般法則：

	乘法 8	乘法16	乘法32	除法 8	除法 16	除法32
op1	☑	☑	☑	☑	☑	☑
op2	☑	☑	☑	☑	☑	☑
op3	-	☑	☑	-	☑	☑
op4	-	-	☑	-	-	☑
tmp1	-	-	-	-	-	☑

● 加減法運算：視運算式的複雜度而有不同的暫存器調整，實際暫存器運用情況請查看*.s 檔的轉譯內容。

	減法 8	減法16	減法32	加法 16	加法32
op1	-	☑	☑	☑	☑
op2	-	☑	☑	☑	☑
op3	-	-	-	-	-
op4	-	-	-	-	-
tmp1	-	-	-	-	-

	記憶體內容複製	指標（讀取/寫入）運算
op1	☑	☑
op2	☑	☑
op3	☑	☑
op4	-	-
tmp1	-	-

附註：以下試舉三例，其會引發記憶體內容複製的運算：

1. 字串初始化
2. Table ROM 複製
3. 字串運算

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 為結構或聯合中成員的一系列的宣告。

結構與聯合的差異，由定義語法來看是相似的；但以記憶體的觀點來看的話，struct 的各成員各自有自己的記憶體空間，一個 struct 佔用空間是 struct 各成員所占記憶體大小的總合，再加上邊界對齊（boundary alignment）。

不同於結構，union 不為每一個資料成員配置空間，而是所有 union 資料成員共用同一個記憶體空間。其空間的大小為成員之中最大資料類型的空間；因此，在 union 某一欄位的資料內容有任何異動時，皆會相對應地影響到其它欄位之內容。意即同時間只能儲存其中一個成員的資料。故而，一個聯合只配置一個足夠大的空間，來容納最大長度的資料成員。

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.
*/
```

結構、聯合中宣告和使用位元欄位

ANSI C 允許整數成員儲存在小於編譯器所允許的記憶體空間。此節省空間的結構成員稱為位元欄位（**bit fields**），而且可以明確宣告其位元長度。在程式中使用位元欄位，必須限定資料結構固定對應到一硬體屬性，因此不太可能具移植性（unportable）。

無論在應用中是定義 signed int 或 unsigned int，成員的預設資料型態為 **int**。所以，在宣告的同時，您必須定義 bit fields 的符號。

```
struct on_off {
    unsigned light : 1;//低位元
    unsigned toaster : 1;
    unsigned ac : 4;
    unsigned clock : 1;
    unsigned flag: 1;//高位元
} kitchen ;
```

以上範例，結構 kitchen 包含五個成員總共為 1 byte。以下表格說明瞭每個成員所佔據的空間：

成員名稱	占據空間
light	1 bit
toaster	1 bit
ac	4 bits
clock	1 bit
flag	1 bit

在前一節中提到，在 union 某一欄位的資料內容有任何異動時，皆會相對應地影響到其它欄位之內容。如此的 union 特性，在硬體程式設計的應用中提供了一個有效率的儲存空間控制。以下舉例說明：

假設程式碼中定義了一變數 flag，占一個 byte，在實作上同時想對 flag 的每一個位進行各別設定時，就可以通過 union 的定義來對占相同記憶體空間的 Byte 與 bit 資料成員進行操作。

```
union test{
    unsigned char flag;
    bit flag_bit0; //低位元
    bit flag_bit1;
    bit flag_bit2;
    bit flag_bit3;
    bit flag_bit4;
    bit flag_bit5;
    bit flag_bit6;
    bit flag_bit7;//高位元
}TEST;
```

通過 TEST 實例的操作，我們可以同時對一個 byte 資料進行操作（例如，TEST.flag=10;），也可以對資料的各個 bit 進行操作（例如，TEST.flag_bit=1;）。如此，不用通過 or、and 或 shift 運算，即可以實現相同的效果。這在 8-bit 單片機的程式設計中，提供了 C 程式碼的直覺性與便利性，又同時兼顧組合語言的靈活性。

註：

1. 因結構中各個結構成員之資料型態不限定；當 bit-field 與其它資料型態成員同時宣告在同一結構時，為節省位址空間，請盡量連續宣告 bit-field 為宜，如此鏈結器將會以 byte 為單位元集中配置位元址。
2. 結構和聯合也可做指定位址的宣告。以下為指定位址之語法：

declaration:	*struct-or-union identifier Var_name @ Address* :(**RPLANE**|**FPLANE**)

位元資料型態

有別於 ANSI C，TM57 C 語言編譯器內建支援位元欄位（bit fields）。用位元資料型態以儲存布林資訊，例如 1 或 0（真或假）。使用位元資料型態來代表資料的真/假（或是/否），例如，狀態標誌，LED 狀態 ... 等。

在 TM57 單晶片系列，控制暫存器的基本資料單位為二進制數字或位元。超過兩個狀態的值需要多個位元元來表示。當宣告一個定址在 F-Plane 的位元變數時，請注意 F-Plane 的前半段是可位元定址的，而後半段是不可位元定址的，詳細請參考規格書。其表示方式如下：

bit-specifier:	**bit** *identifier* **bit** *idenifier@address@bit***:***plane_type* **bit** *idenifier***:bank1**
plane_type	**RPLANE** **FPLANE**

注意：只能在**全域區域（global area）**中，宣告位元型態變數。

範例：宣告由 link 自動分配位址之全域位元變數。

```
const int gInt = 10;
bit bit1;
bit bit2:bank1;
main()
{
   int kk;
   bit1 = 1;
   bit2 = 0;
   kk = gInt;
}
```

Var name	Hex Value	Addr/PC Range	Data Type	Bank no.	Plane
D:\bcb_test\pa20_0720\DEBUG...					
Global Var					
D:\bcb_test\pa20_0720\DE...					
gInt	000A				
bit1	0	0020.0	bit	Bnk0	FPlane
bit2	0	0028.0	bit	Bnk1	FPlane
Auto Var					
D:\bcb_test\pa20_0720\DE...					
main		PC: 0001~0009			
kk	531F	0021~0022	int	Bnk0	FPlane

Address: 0020.0 (0x0020 bit 0)

範例：宣告一全域位元變數並且指定位址在 F-PLANE。

```
bit gBit_1 @0x20@0:FPLANE;
bit gBit_2 @0x20@1:FPLANE;
bit gBit_3 @0x20@2:FPLANE;

main()
{
    gBit_1 = 0;
    gBit_2 = 1;
    gBit_3 = 0;
}
```

Address: 0x0020 in F-PLANE

0 | 1 | 0

Bit 2 : gBit_3
Bit 1 : gBit_2
Bit 0 : gBit_1

Var Name	Hex Value	Addr/PC Ran...	Data Type	Bank no.	Plane	File ...
D:\bcb_test\c_switc...						
Global Var						
D:\bcb_test\c...						
gBit_1	0	0020.0	bit	Bnk0	FPlane	D:\...
gBit_2	1	0020.1	bit	Bnk0	FPlane	D:\...
gBit_3	0	0020.2	bit	Bnk0	FPlane	D:\...
Auto Var						
D:\bcb_test\c...						

C Variables | ASM F_Plane 1 | ASM F_Plane 2 | ASM R_Plane

位元變數可依實際需求以指定定址在 F-Plane 的 bank1（當未指定時，鏈結器預設定址位址在 bank0，故而無需指定定址在 bank0）。鏈結器將定址該位元變數在 F-Plane 的 bank1 之最開始的位址。以下圖 TM57FLA80 為例，bank1 可定址之起始位址為 0x30，所以指定定址在 bank1 的位元變數 b0、b1、b2、b3 其位址為 0x30 的第 0~3 個 位元。

請注意該例子之長整數變數：l1，其儲存位址為跨越 bank0 及 bank1，鏈結器將配置變數 l1 位元址為 bank0 的 0x7e~0x7f 及 bank1 的 0x31~0x32，避開已配置之位元元位元址 0x30。意指當有位元變數指定定址在 bank1 及程式中有變數其儲存位址為跨越 bank0 及 bank1 時，跨越 bank 的變數其配置位元址將由鏈結器自動調整設定，以避免重覆定址。

```
1  unsigned int test[47];
2  long l1;
3  char c1;
4  bit b0:bank1;
5  bit b1:bank1;
6  bit b2:bank1;
7  bit b3:bank1;
8  main()
9  {
10     l1=0x12345678;
11     c1=0x20;
12     b0=1;
13 }
14
```

13 : 1 | Insert | Single Step ... | Connecting ... | TM57FLA80

Var Name	Hex Value	Addr/PC Range	Data Type	Bank no.	Plane	File Name
D:\bcb_test\c_s...						
Global Var						
D:\bcb_tes...						
test		0020~007d	UINT[47]	Bnk0	FPlane	D:\bcb...
l1	12345678	007e~007f, 0031~0032	long	Bnk0,Bnk1	FPlane	D:\bcb...
c1	20 ' '	0033~0033	char	Bnk1	FPlane	D:\bcb...
b0	1	0030.0	bit	Bnk1	FPlane	D:\bcb...
b1	0	0030.1	bit	Bnk1	FPlane	D:\bcb...
b2	0	0030.2	bit	Bnk1	FPlane	D:\bcb...
b3	1	0030.3	bit	Bnk1	FPlane	D:\bcb...

建議：宣告位元變數時，請盡量連續宣告為宜，如此鏈結器將會集中配置位元址，以節省位址空間。反之，若非連續宣告位元變數，鏈結器將不會連續配置位元址，將造成位址空間之浪費。

位元運算元

針對位元的資料型態，TM57 C 語言編譯器支援以下運運算元：

- 數學運運算元：+ - * /
- 位元元方式運運算元：& | ~

注意：不支援位元變數的位移（<< >>）

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：

- 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。
- 若沒有給定一明確的值，列表最左邊的常數預設為零（0）。
- 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 TM57 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;
```

注意：不支援常數指標（constant pointer）宣告

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

TM57 C 語言編譯器作用於 8-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	1	0 ~ 255
bit	1	0 ~ 1

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的型態是可相容的（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
- 傳遞變數的位址到一函式。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。索引操作符（[ ]）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，TM57 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";
static char name4[4]= "Joys";//Error 因最後結尾包含\0，最多只能放 3 個字元
```

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，若二者的大小表示符為常整數運算式，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 ）

TM57 C 語言函數的回傳值將被存在 F-Plane RAM，即存在運算寄存器 op2。函數原型在函式呼叫前必須先定義。此外，如同 const 變數， 亦允許使用者定義函式在 TABLE ROM (program memory)的地址。如以下例子， 函式 f1 指定 ROM 地址 0x0100，而函式 f2 不指定 ROM 地址。請注意，只能在函式定義區段中指定 ROM 地址。

```
char f1(char, char);            // 無法在函式中指定地址
char f2(char, char);
void main()
{
char a = 5;
char b = 2;
char c = 0;
c = f1(a, b);
c = f2(a, b);
}
char f1(char i, char j)@0x0100     //在函式定義中指定ROM地址0x0100,
{
if(i < j)
return (j - i);
else
return (i - j);
}
char f2(char i, char j)
{
return (i + j);
}
```

asm 宣告

關鍵字 **asm** 指的是組合語言（assembly code）。在實行的過程中，TM57 系列 C 語言編譯器將認得宣告式的關鍵字 asm 但會忽略之。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。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 TM57XX 操作碼（opcodes）用在內部組譯器。

內建的內部組譯器並不是要取代巨集組譯器。

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，運算式子會插入一些格式符號。

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）參數或函式的組合器名稱
%o	（區域）參數的堆疊偏移量
%%	% 符號

%n	專門針對 bsf、bcf 、btfsc、btfss 指令，若為上述指令時，會去判斷 bit 變數在該哪一個位元，並加入到指令當中。
	※ 只適用於 bit 型態的變數
	Ex:
	bit bb;//假設在 0x20, bit 0 的位置
	asm(“bsf %n”,bb); //
	
	結果:
	bsf 0x20,0

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 W 累加器，可用以下寫法：

```
#define OFFS  23
__asm__ ("MOVLW %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__ ("MOVLW %v+%b", pixel,offsetof(pixel_t, color));
```

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__ ("MOVFW _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：

```
MOVLW $00 ;1,n=1,2
MOVWF _foo+0
MOVWF _foo+1
```

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__(“MOVFW %v”, 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 *tar,char *src)
{
    // Return tar value from op2 (0x24)
    asm("movfw %o",tar);
    asm("movwf op2");        // return tar pointer in op2 address (0x24)
    asm("_strcpy_LOOP:");  // generate label name
    // Read from source
    asm("movfw %o",src);              // Set offset of LOCAL name src
    asm("call runtime_Ind_Read");    // call indirect read
    asm("movwf op3");                 // op3 to write to target
    // Save to target
    asm("movfw %o",tar);              // Set offset of LOCAL name tar
    // (strcpy_LOCAL+1)
    asm("call runtime_Ind_Write");    // call indirect write
    // Check end
    asm("testz op3");
    asm("btfsc STATUS, ZERO_FLAG");
    asm("ret");
    // Next
    asm("incf %o,1",src);
    asm("incf %o,1",tar);
    asm("goto _strcpy_LOOP");
}
```

範例 2：在*.C 檔案呼叫 ASM

```
main(void)
{
   asm(“call  asmLabelDelay”);      // in *.asm file need “.exportasm LabelDelay”
}
```

在附錄章節的例子 1 中，列示一連串以 inline asm 來實作字串的相關運算函式。

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;
}
```

- 雖然指標可以指向函式的矩陣，但是不允許函式的矩陣。結構或聯合不能包含函式。以下結構宣告的範例是不合規定的：

```
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 宣告並沒有預留其記憶空間。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）

利用物件或未知大小的矩陣的宣告，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）

TM57 C 語言編譯器，可對 *struct* 或 *union* 類別物件進行初始化，即使他們有自動儲存持續期間。*unions* 是利用第一個宣告的元素的類別以進行初始化。當參數被宣告為 *struct* 或 *array*，初始化式子包含一個由括弧包圍住並以逗號隔開的初始列，來對以遞增的索引符號或依成員的順序之聚集的成員進行初始化。

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";
```

3. 算式和運算符號（Expressions and Operators）

此章節介紹在 C 語言中，可用的各種運算式和運運算元。此章節的運算式和運運算元大約按照其優先序以說明。

在 C 語言的運算符號優先序和結合律規則

C 語言的運運算元有優先序和結合律規則，其用以決定運算式如何將運運算元做組合且算出結果。優先序是指對不同類型的運運算元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運算元和運算式，其“由左至右”或“由右至左”的結合順序。運運算元的優先權是具意義的，尤其在當有其他具更高或更低優先權的運運算元同時存在時。運算式中具更高優先權的運運算元將被優先處理。

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

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

主要的式子（Primary Expressions）

以下列示的式子皆為“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，藉由索引符號做修改
以括弧包住的式子	含括弧的式子其數值與不含括弧的式子是相同的。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）

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

運運算元的功能	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++

矩陣索引符號運運算元（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索引符號參考。矩陣的第一個元素索引符號為 0。句子 code[10]指的是矩陣的第 11 個元素。

在多維矩陣中，最常用的方式是以增加最右邊索引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/聯合成員的值，而且若第一個句子為左值（lvalue），它也必須為左值。運算結果含有結構/聯合指定成員的型態與結構/聯合的限定詞。

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

透過 ->（箭頭）運運算元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。

在後置 ++ 和 -- 的運運算元中，運算式儲存值的更新時間點，將被延誤到下一個序列點。

單元運運算元（Unary Operators）

單元運算式包含一個運算元和一個單元運運算元。所有單元運運算元具有相同優先權與***由右至左的結合律***。因此單元運運算元為後置運算式。如下麵的說明，算術轉換通常是在單元式子的運算式中進行的。以下表格總結會出現在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

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

單元運運算元 * 意指間接引用；型態轉換式必須是一個指標，而且其結果值等同於運算式所指向的物件之左值或者一個函式的指定符。單元運運算元 & 的運算元可以是函式指定符或是指向物件的左值。單元運運算元 & 的運算結果是一左值，其為用來指向一物件的指標，或是一函式指定符所參考的函式。

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

單元運運算元 - 的結果值是運算元的負數。在運算式中將進行整數四捨五入，而且其結果是四捨五入後的類別和運算式的負數值。

單元運運算元 + 維持運算式的值。運算元可以是算術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 ~ 運運算元

邏輯否定運運算元！決定運算元運算結果為 0（假）或非零（真）。若運算元的值為 0 則經邏輯否定運運算元！運算後的結果值為 1，反之，若運算元的值為非零則結果值為 0。

以下兩個式子的運算結果是相同的：

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

位元否定運運算元 ~ 產生運算元的位元元方式之補數。結果以二進制表示，在二進制運算式的標記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 8 位元二進制表示 x 為：00000101。式子~x 產生的結果是：11111010

前置 ++ 和 -- 運運算元

前置運運算元 ++ 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運算元 ++ 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 單元運運算元

sizeof 運運算元可得到運算式的大小，以 bytes 為單位。其可以是具資料型態的運算式或有括弧的資料型態名稱。

在 TM57 C 語言編譯器，一個字元 *char* 的大小為 1，一個整數 *int* 的大小為 2，一個長整數 *long* 的大小為 4。其主要用在程式的溝通，例如配置儲存體和 I/O 系統。sizeof 運運算元的語法如下：

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

sizeof 運運算元**不能**應用於：

- 位元域
- 函式類別
- 未定義的結構或類別
- 不完整的類別（如 void）

乘法運運算元（Multiplicative Operators）

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

```
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。二元運運算元 * 意指乘法，其結果是運算元的乘積。二元運運算元 / 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，商數其數值大小為小於或等於正商數。而且具相同的正負號。

二元運運算元 % 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。

加法運運算元（Additive Operators）

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

```
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，其索引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;
```

位移運運算元（Shift Operators）

位移運運算元以位元元方式來移動二進制物件的位元值。位元元方式位移運運算元 << 和 >> 具“由左至右” 結合性。每個運算元必須為整數型態。每個運算式都應用整數四捨五入。其語法如下：

```
shift-expression:    additive-expression
                     shift-expression << additive-expression
                     shift-expression >> additive-expression
```

運算符號	功能
<<	表示位元將被移至左邊
>>	表示位元將被移至右邊

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

0000111110110011

式子 left_op << 3 結果為：

0111110110011000

關係運運算元（< > <= >=）

關係運運算元對兩個運算元作比較並且決定此關係的正確性。其結果的型態為 int，結果值為 1 則定義關係為真（true）；反之，結果值為 0 則關係為假（false）。其結果並非為左值。

運算符號	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。

相等運運算元（== !=）

如同關係運運算元，相等運運算元為比較兩個運算元來決定其關係的正確性。然而，相等運運算元的優先權低於關係運運算元。其結果的型態為 int，若定義的關係為真（true）則數值為 1，反之，則數值為 0。

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

邏輯運運算元 AND（&&），邏輯運運算元 OR（||）

邏輯運運算元 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，其結果值為 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

邏輯運運算元 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 也會對每個運算元進行一般的算術轉換。

<table>
<tr><th>句子</th><th>結果</th></tr>
<tr><td>1 || 0</td><td>1</td></tr>
<tr><td>1 || 6</td><td>1</td></tr>
<tr><td>0 || 0</td><td>0</td></tr>
</table>

條件運運算元（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，計算條件運算式並判斷其結果值，若值為真則計算 $operand_2$；反之，則計算 $operand_3$。

```
( operand1 ? operand2 : operand3 )
```

計算第一個運算元並以其結果值，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

條件運算式的結果為第二或第三運算式的值。

以下句子決定哪一個參數為比較大的值，y 或 z，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;
```

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，述句將依序一一被執行。

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。有一個特例是只包含一個分號的空述句。

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。在複合述句中的宣告為具有區域作用域（block scope）。意即在宣告列中的識別符號，若曾被外部宣告過，則外部宣告在此區塊的運行期間是被隱藏的，直到它獲得焦點後才恢復。

選擇述句（Selection Statements）

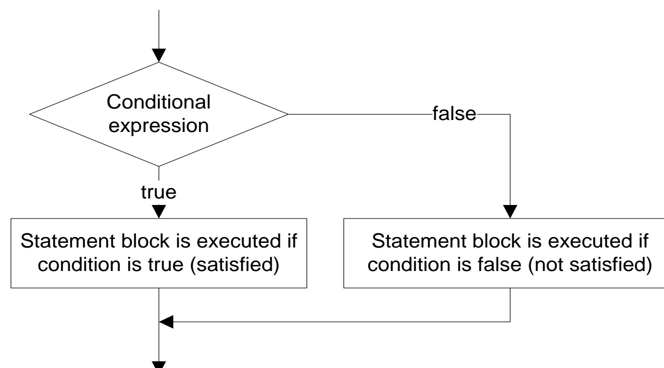
選擇述句包含 if 和 switch 述句。選擇述句將根據算式的判斷，以選擇其中一項述句來執行。***Expression*** 為控制運算式。

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，來轉移控制到不同的述句。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
- 隨意的 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;
}
```

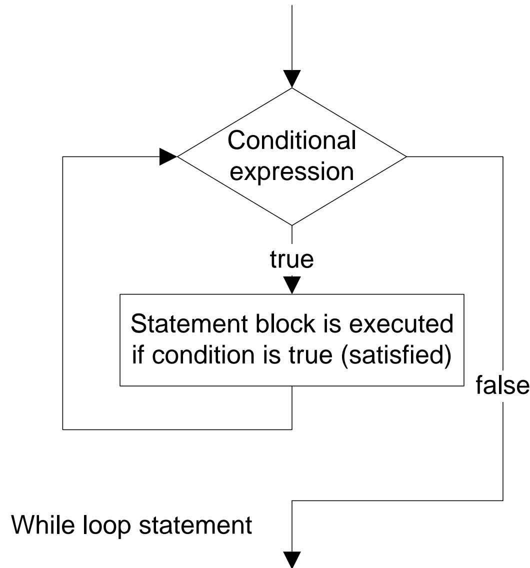
重複述句

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 ([$expression_1$] ; [$expression_2$] ; [$expression_3$]) statement

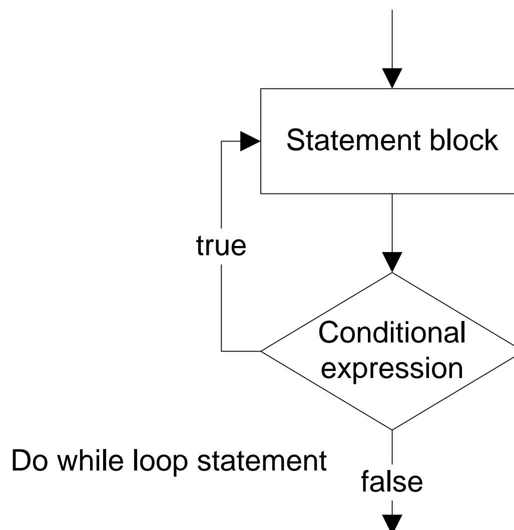
while 述句

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



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。



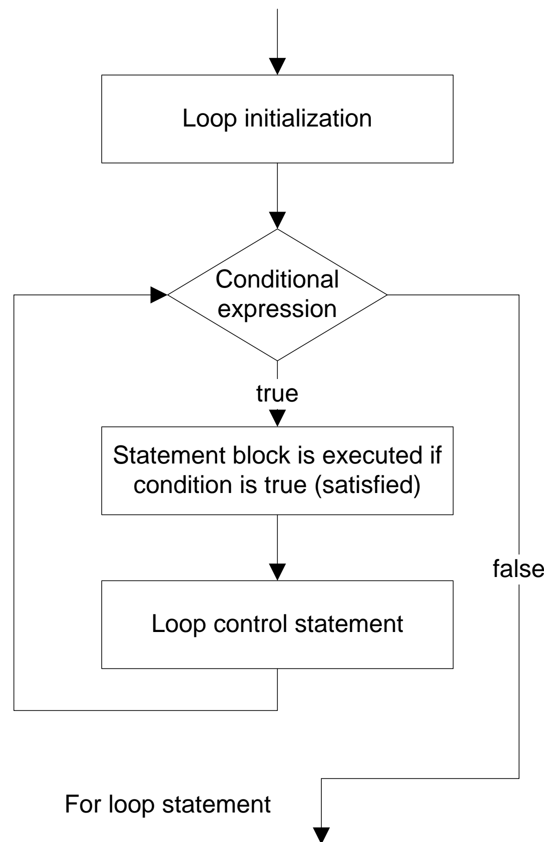
for 述句

for 述句讓您作以下的事：

- 在執行第一次述句（$expression_1$）前，先判斷述句（初始化）。
- 指明一運算式（$expression_2$）以判斷述句是否該進行（依條件）。
- 在每次述句的反覆運算後，執行運算式 $expression_3$（常用來遞增每次的重複）。
- 若控制算式（$expression_2$）不是 0，則重複處理述句。

即便第二個算式不是為 0，**break**、**return** 或 **goto** 述句亦可讓 for 述句終止。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;

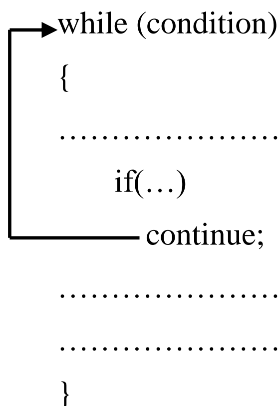
識別符號必須命名為處於封閉式函式內的標籤。若標籤名稱未出現，將被視為隱藏式的宣告。因為 goto 述句會幹擾正常的流程序列，造成程式變得更難閱讀和維護。建議以 break 述句、continue 述句或函式呼叫來取代 goto 述句的需求。

範例：

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

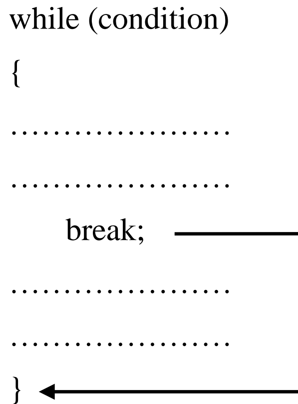
continue 述句

- continue 述句只能出現在重複述句的 body。continue 述句會導致控制忽略循環的後續部分，直接到達循環的端點，即形成最小的封閉 while、do、for 述句。



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，並由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其控制到迴圈以外的下一個述句。



return 述句

return 述句終止目前執行的函式，並且回傳執行控制權給函式的呼叫者。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 TM57 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> 意指在 MCU 中支援的中斷向量，例如在 TM57FLA80 晶片，會有以下這些中斷：Pin interrupts、Timer interrupt、PWM/CMP/ADC 和 Wakeup Timer/USI interrupt。我們提供序列 0x01(TMR0)、0x02(TMR1)、0x03(TMR2)、0x04(PWM0)、0x05(WKT)、0x06(XINTA)、0x07(XINTB)、0x08(UART)、0x09(SPI)來符合 IC 中斷向量。

TM57FLA80 晶片：中斷的宣告如下：

```
void interrupt TMR0_Intrerrupt(void) @ 0x01 {….}
void interrupt TMR1_Intrerrupt(void) @ 0x02 {….}
void interrupt TMR2_Intrerrupt(void) @ 0x03 {….}
void interrupt PWM0_Intrerrupt(void) @ 0x04 {….}
void interrupt WKT_Intrerrupt(void) @ 0x05 {….}
void interrupt XINTA_Intrerrupt(void) @ 0x06 {….}
void interrupt XINTB_Intrerrupt(void) @ 0x07 {….}
void interrupt UART_Intrerrupt(void) @ 0x08 {….}
void interrupt SPI_Intrerrupt(void) @ 0x09 {….}
```

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

範例：

```
//RPLANE
char OPTION @0x02:RPLANE;

//FPLANE
char INTE1 @0x08:FPLANE;
char INTF1 @0x09:FPLANE;
char TIMER0 @0x01:FPLANE;
char TM1CTRL @0x0D:FPLANE;

int temp_b=0;
unsigned char  Timer_Buf=0xF0;
main()
{
        char c=0;
        int a=0;

        TIMER0=Timer_Buf;
        OPTION=0;
        INTE1=0x10;    // Enable Timer0 Interrupt

   for(c=0;;c++)
        ++a;
}
```

```
void interrupt TM0_Interrupt(void)@0x01
{
        TIMER0=Timer_Buf;
        INTF1=0;
        ++temp_b;
}
```

在上述範例，當 TMR0 中斷相關設定已設定，然後用一個無限迴圈讓 TMR0 溢位，以執行中斷服務。

注意：不同於普通函式的用法，只要符合中斷條件，中斷應用將自動被執行。

若在實際應用中，中斷被觸發並且在真正執行中斷服務程式之前，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式執行過程中，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式後能回存運算資料。如此讓控制程式能在中斷後，仍然正確且持續的執行下去，意即確保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）組語副程式已於各 TM57 系列單晶片的 runtime library 中實作出來，使用者可依實際應用來呼叫上述二個副程式。另外，為避免上述二個副程式執行時間過久，額外提供 ISR_SaveData_5、ISR_RestoreData_5 二個組語副程式；ISR_SaveData 與 ISR_SaveData_5 的不同，主要在所暫存的運算資料不同（後面會加以詳述）。使用者可依實際運算的複雜程度，以決定應使用何種中斷保護副程式來有效率的儲存 Data RAM 資料。

為了不影響現有既存的記憶體資料內容，在中斷產生時，ISR_SaveData 及 ISR_SaveData_5 程式將需預先暫存的運算資料，存放在**記憶體最後面的位址**中。鏈結器（linker）是依整個專案及設定連結的函式庫程式，整體性地計算其所有程式所使用到的運算暫存器或堆疊指標暫存器之數量。並據此數量自最後位址開始計算出所需之位址空間，依照前後順序存放工作暫存器、狀態暫存器、op1~op4 或 stkptr 等資料內容在記憶體的末端。

資料暫存的記憶體位置可依單晶片特性區分為三種：(1) R-Plane、(2) F-Plane Bank0、(3) F-Plane Bank1。以下將依據 TM57 系列各晶片特性分別陳述其實作細節。

(一) R-Plane

狀態值及運算資料儲存在 R-Plane RAM 記憶體最後面的位址。以單晶片 TM57FLA80 為例，假設需要儲存的運算資料包含：工作暫存器（working register）、狀態暫存器（status register）及 op1，則資料的儲存位址分別為：

	儲存位址
工作暫存器	0xFA
狀態暫存器	0xFB
op1	0xFC~0xFF

資料存在 R-Plane 的 TM57 系列晶片為下列，其共同特性為

- R-Plane 記憶體具可讀可寫。

Rplane 最終位址	
TM57FE80	0xFF
TM57FLA80	0xFF
TM57ML40	0xFF

(二) F-Plane Bank 0

狀態值及運算資料儲存在 F-Plane Bank0 記憶體的最後位址。以單晶片 TM57PE11 為例，假設需要儲存的運算資料包含：工作暫存器、狀態暫存器及 op1，則資料的儲存位址分別為：

	儲存位址
工作暫存器	0x4A
狀態暫存器	0x4B
op1	0x4C~0x4F

資料存在 F-Plane Bank0 的 TM57 系列晶片為下列，其共同特性為

- F-Plane 只有單一個 Bank。
- R-Plane 只允許寫入但不允許讀取。

Fplane bank0 最終位址	
TM57ME20	0x7F
TM57P11	0x4F
TM57P12	0x4F
TM57PA10	0x5F
TM57PA10A	0x5F
TM57PE10	0x4F
TM57PE11	0x4F
TM57PE11A	0x4F
TM57PE12	0x4F
TM57PE12A	0x4F
TM57RE12	0x4F

(三) F-Plane Bank 1

狀態值及運算資料儲存在 F-Plane Bank 1 記憶體的最後位址。以單晶片 TM57PA40 為例，假設需要儲存的運算資料包含：工作暫存器、狀態暫存器及 op1，則資料的儲存位址分別為：

	儲存位址
工作暫存器	0x7A
狀態暫存器	0x7B
op1	0x7C~0x7F

資料存在 F-Plane Bank 1 的 TM57 系列晶片為下列，其共同特性為

- F-Plane 有二個 Bank。
- R-Plane 只允許寫入但不允許讀取。

F-Plane bank1 最終位址	
TM56FA40	0x7F
TM57FA40	0x7F
TM57PA40	0x7F
TM57PA20	0x7F
TM57PA20A	0x7F
TM57PE40	0x7F
TMU3130	0xFF
TMU3131	0xFF
TMU3132	0xFF

注意：雖然 TMU3130、TMU3132 及 TMU3132 的 R-Plane 記憶體可讀可寫，但與 USB 讀寫位址衝突。故而將運算資料儲存位置改為 F-Plane Bank1。

ISR_SaveData、ISR_RestoreData

ISR_SaveData 所儲存的暫存器內容，是所有程式中**運算所需的至多暫存器內容**（而**非觸發中斷當時已使用到的暫存器**）。故而，為了更有效率的運用有限的記憶體空間，使用者可自行依據實際中斷應用另寫儲存/回存函式，或使用僅儲存重要暫存器內容的 ISR_SaveData_5 函式，以真實回應暫存器的使用情形。

假設在觸發中斷時，程式只使用到 op1~op2，但整個專案程式至多會使用到 op1~op4。執行 runtime library 的 ISR_SaveData()將儲存工作暫存器、狀態暫存器及 op1~op4，使用者可自行定義類似 ISR_SaveData()的函式（或使用 ISR_SaveData_5）來儲存工作暫存器、狀態暫存器及 op1~op2，以節省記憶體空間。當自訂儲存及回存函式時，需提醒使用者在編碼時注意以下規則：

- 請遵循單晶片的特性並參考上述各晶片適用情況，決定要暫存的記憶體位置，例如 TM57FA40 就應存在 F-Plane 的 Bank1；TM57FLA80 就應存在 R-Plane。
- 狀態及運算資料存放在 R-Plane 及 F-Plane 的 Bank0 時，儲存及回存資料無需考量 Bank 之間切換並複製資料的問題。唯有存在 F-Plane 的 Bank1 時，需要注意此問題（因為，狀態及運算暫存器是預設存放在 F-Plane 的 Bank0）。詳細的實作細節請參考以下的範例解說。

以下範例為觸發中斷及進入中斷服務程式之 C 程式，其呼叫 ISR_SaveData()及 ISR_RestoreData()以儲存及回存運算相關 Data RAM 資料，供使用者參考（註，在 C 程式中呼叫 ISR_SaveData_5 及 ISR_RestoreData_5 為相同的方式）。

C 程式：counter_Function() 為實際執行之中斷服務程式。

```
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
{
       ….
       ISR_SaveData();       // Save data
       counter_Function();
       ISR_RestoreData();   // Restore data
       ….
}
// Initial process
void  predivider_initial(void)
{
       ….. // initialization process
}
void  counter_Function(void)
{
     ….. // do something
}
```

以下為實作在 runtime library 中 ISR_SaveData 及 ISR_RestoreData 儲存及回存運算相關 Data RAM 資料之組語程式，分為三種類型：R-Plane、F-Plane Bank0 及 F-Plane Bank1 來舉實例說明：

- **R-Plane**：以下舉 TM57FLA80 為例

```
.autoimport on
.export _ISR_SaveData,_ISR_RestoreData

 SAVEADDR = __RPLANE_ADDRESS_MAX__-( 1 +__OPXSTKPTRSIZE__)

.proc _ISR_SaveData

    MOVWR SAVEADDR
    MOVFW STATUS
    MOVWR SAVEADDR+1

    MOVLW __OPXSTKPTRSIZE__
    IORLW 0
    BTFSC STATUS,ZERO_FLAG
    RET

    MOVFW op1
    MOVWR SAVEADDR+2
```

```
    MOVLW op1+1
    MOVWF FSR
    MOVLW SAVEADDR+3
    MOVWF RSR
    MOVLW __OPXSTKPTRSIZE__ - 1
    CALL  runtime_ISR_data_copy_F2R
    RET

.endproc

.proc _ISR_RestoreData

    MOVLW __OPXSTKPTRSIZE__
    IORLW 0
    BTFSC STATUS,ZERO_FLAG
    GOTO W_S

    MOVLW SAVEADDR+3
    MOVWF RSR
    MOVLW op1+1
    MOVWF FSR
    MOVLW __OPXSTKPTRSIZE__ - 1
    CALL  runtime_ISR_data_copy_R2F

    MOVRW SAVEADDR+2
    MOVWF op1

W_S:
    MOVRW SAVEADDR+1
    MOVWF STATUS
    MOVRW SAVEADDR

RET

.endproc

.export runtime_ISR_data_copy_F2R
.proc runtime_ISR_data_copy_F2R

        movwf op1 ; counter
        addwf FSR,1
        addwf RSR,1

loop:
        decf RSR,1
        decf FSR,1
        movfw R0
        MOVWR R0
        decf op1
        testz op1
        btfss STATUS,ZERO_FLAG
        goto loop

ret
```

```
    .endproc

        .export runtime_ISR_data_copy_R2F
        .proc runtime_ISR_data_copy_R2F

                movwf op1 ; counter
                addwf FSR,1
                addwf RSR,1
    loop:
                decf op1,1
                decf FSR,1
                decf RSR,1

                MOVRW R0
                movwf R0

                testz op1
                btfss STATUS,ZERO_FLAG
                goto loop
    ret

.endproc
```

- **F-Plane Bank0**：以下舉 TM57PA10 為例

```
.autoimport on
.export _ISR_SaveData,_ISR_RestoreData

#define _BANK_FLAG  5
  SAVEADDR = __FPLANE_ADDRESS_MAX__-(1+__OPXSTKPTRSIZE__)

.proc _ISR_SaveData

    MOVWF SAVEADDR
    MOVFW STATUS
    MOVWF SAVEADDR+1

    MOVLW __OPXSTKPTRSIZE__
    IORLW 0
    BTFSC STATUS,ZERO_FLAG
    RET

    MOVFW op1
    MOVWF SAVEADDR+2
    MOVFW op1+1
    MOVWF SAVEADDR+3
    MOVFW op1+2
    MOVWF SAVEADDR+4
    MOVFW op1+3
    MOVWF SAVEADDR+5

    MOVLW __OPXSTKPTRSIZE__
    XORLW 4
    BTFSC STATUS,ZERO_FLAG
    RET
```

```
    MOVLW op2
    MOVWF op1
    MOVLW SAVEADDR+6
    MOVWF op1+1
    MOVLW __OPXSTKPTRSIZE__ - 4
    CALL  runtime_ISR_data_copy
    RET

.endproc

.proc _ISR_RestoreData

    MOVLW __OPXSTKPTRSIZE__
    IORLW 0
    BTFSC STATUS,ZERO_FLAG
    GOTO W_S

    MOVLW __OPXSTKPTRSIZE__
    XORLW 4
    BTFSC STATUS,ZERO_FLAG
    GOTO  RES_OP1

    MOVLW SAVEADDR+6
    MOVWF op1
    MOVLW op2
    MOVWF op1+1
    MOVLW __OPXSTKPTRSIZE__ - 4
    CALL  runtime_ISR_data_copy

RES_OP1:
    MOVFW SAVEADDR+2
    MOVWF op1
    MOVFW SAVEADDR+3
    MOVWF op1+1
    MOVFW SAVEADDR+4
    MOVWF op1+2
    MOVFW SAVEADDR+5
    MOVWF op1+3

W_S:

    MOVFW SAVEADDR+1
    MOVWF STATUS
    MOVFW SAVEADDR
    RET

.endproc

.proc runtime_ISR_data_copy

            movwf op1+2 ; counter
loop:
            decf op1+2,1

            ; Source from RAM ( F plane )
```

```
            ;CALL runtime_Ind_bank_switch_2
            movfw op1
            movwf FSR
            movfw R0
            movwf op1+3

            movfw op1+1
            movwf FSR

            movfw op1+3
            movwf R0

        testz op1+2
        btfsc STATUS,ZERO_FLAG
        ret
        incf op1,1
        incf op1+1,1
        goto loop

.endproc
```

- **F-Plane Bank1**：以下舉 TM57PA40 為例

```
.autoimport on
.export _ISR_SaveData,_ISR_RestoreData

#define _BANK_FLAG  5
SAVEADDR = __FPLANE_ADDRESS_MAX__-(__BANK1_OFFSET_VALUE__ -
0x80+1+__OPXSTKPTRSIZE__)

.proc _ISR_SaveData

        BTFSC STATUS,_BANK_FLAG
        GOTO BANK1
        MOVWF SAVEADDR
        BCF STATUS,_BANK_FLAG
        goto B0

BANK1:
        MOVWF SAVEADDR

B0:
        CALL __DUMMY_FUNC__
        MOVFW STATUS
        MOVWF SAVEADDR+1

        MOVLW __OPXSTKPTRSIZE__
        IORLW 0
        BTFSC STATUS,ZERO_FLAG
        RET

        MOVFW op1
        MOVWF SAVEADDR+2
        MOVFW op1+1
```

```
        MOVWF SAVEADDR+3
        MOVFW op1+2
        MOVWF SAVEADDR+4
        MOVFW op1+3
        MOVWF SAVEADDR+5

        MOVLW __OPXSTKPTRSIZE__
        XORLW 4
        BTFSC STATUS,ZERO_FLAG
        RET

        MOVLW op2
        MOVWF op1
        MOVLW SAVEADDR+6
        MOVWF op1+1
        MOVLW __OPXSTKPTRSIZE__ - 4
        CALL  runtime_ISR_data_copy
        RET
.endproc

.proc _ISR_RestoreData
        MOVLW __OPXSTKPTRSIZE__
        IORLW 0
        BTFSC STATUS,ZERO_FLAG
        GOTO W_S

        MOVLW __OPXSTKPTRSIZE__
        XORLW 4
        BTFSC STATUS,ZERO_FLAG
        GOTO  RES_OP1

        MOVLW SAVEADDR+6
        MOVWF op1
        MOVLW op2
        MOVWF op1+1
        MOVLW __OPXSTKPTRSIZE__ - 4
        CALL  runtime_ISR_data_copy

RES_OP1:
        MOVFW SAVEADDR+2
        MOVWF op1
        MOVFW SAVEADDR+3
        MOVWF op1+1
        MOVFW SAVEADDR+4
        MOVWF op1+2
        MOVFW SAVEADDR+5
        MOVWF op1+3

W_S:
        MOVFW SAVEADDR+1
        MOVWF STATUS
        BTFSS STATUS,_BANK_FLAG
        GOTO L0
        MOVFW SAVEADDR
        RET
```

```
L0:
        BSF STATUS,_BANK_FLAG
        MOVFW SAVEADDR
        BCF STATUS,_BANK_FLAG
        RET

.endproc
.proc __DUMMY_FUNC__

        ret
.endproc
```

以下為 Bank0 與 Bank1 之間資料複製的程式。

```
.autoimport on
.export runtime_ISR_data_copy

.proc runtime_ISR_data_copy

        movwf op1+2       ; counter

loop:
        decf op1+2,1

        ; Source from RAM ( F plane )
        ;CALL runtime_Ind_bank_switch_2
              movfw op1
              btfss op1,7
              goto  L0
              addlw __BANK1_OFFSET_VALUE__
              BSF STATUS,5
              goto L1
L0:
              BCF STATUS,5
L1:
              movwf FSR
              movfw R0
              movwf op1+3
              movfw op1+1
              btfss op1+1,7
              goto  L2
              addlw __BANK1_OFFSET_VALUE__
              BSF STATUS,5
              goto L3
L2:
              BCF STATUS,5
L3:
              movwf FSR
              movfw op1+3
              movwf R0

              testz op1+2
              btfsc STATUS,ZERO_FLAG
```

```
            ret
            incf op1,1
            incf op1+1,1
            goto loop

    .endproc
```

程式中相關變數之定義說明：

變數	意義
__RPLANE_ADDRESS_MAX__	R-Plane 記憶體最終位址。
__FPLANE_ADDRESS_MAX__	F-Plane 記憶體最終位址。
__OPXSTKPTRSIZE__	鏈結器整體性計算出，該專案及設定連結的函式庫程式所使用到的運算暫存器或堆疊指標暫存器，所佔之記憶體大小。
__BANK1_OFFSET_VALUE__	F-Plane Bank1 相對於 Bank0,Bank1 共同資料區塊之位址的位移量。

注意：在中斷應用時，請留意以下二項限制。

1. 不允許巢狀之中斷程式呼叫（即在未離開某一中斷程式前又進入另一中斷程式去執行）。
2. 若中斷程式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。

ISR_SaveData_5、ISR_RestoreData_5

tenx 某些晶片內建一自動儲存功能，其能在中斷程式執行之前及之後，自動儲存及回存工作暫存器及狀態暫存器的內容，只要將下表所列示的旗標開啟，即可觸發自動儲存功能。

晶片名稱	旗標位址
TM57ME20	RPLANE 0x0B,7
TM57FLA80	RPLANE 0x10,4
TM57ML40	RPLANE 0x10,5
TM57FE80	RPLANE 0x07,5
TMU3130	RPLANE 0x07,5
TMU3131	RPLANE 0x07,5
TM56FA40	RPLANE 0x0B,2

上述之自動儲存功能與中斷保護程式，在所儲存的暫存器內容上是重複的（即重複工作暫存器及狀態暫存器），故而中斷保護程式實作在上述的晶片上，就能以更精簡的方式完成工作。

注意：具有自動儲存功能的晶片在中斷保護程式開始之前，中斷保護程式將觸發自動儲存功能，再進行保護動作。因此，**中斷保護程式運行期間，切勿手動關閉自動儲存功能**，此將造成不完全的保護儲存動作。

以下為實作在 runtime library 中 ISR_SaveData_5 及 ISR_RestoreData_5 儲存及回存運算相關 Data RAM 資料之組語程式。我們將分為是否具自動儲存功能的晶片，其分別儲存資料在 Fplane 及 Rplane 來陳列實作程式。

- **R-Plane：**

<table>
<tr><th>有自動儲存功能</th><th>無自動儲存功能</th></tr>
<tr><td><pre>
.autoimport on
.export _ISR_SaveData_5,_ISR_RestoreData_5

 SAVEADDR5 = __RPLANE_ADDRESS_MAX__-(1 + 1)

.proc _ISR_SaveData_5
 MOVFW op1
 MOVWR SAVEADDR5
 MOVFW op1+1
 MOVWR SAVEADDR5+1
 MOVFW op3+2
 MOVWR SAVEADDR5+2
 RET
.endproc

.proc _ISR_RestoreData_5
 MOVRW SAVEADDR5
 MOVWF op1
 MOVRW SAVEADDR5+1
 MOVWF op1+1
 MOVRW SAVEADDR5+2
 MOVWF op3+2
 RET

.endproc
</pre></td><td><pre>
.autoimport on
.export _ISR_SaveData_5,_ISR_RestoreData_5

 SAVEADDR5 = __RPLANE_ADDRESS_MAX__-(1 + 3)

.proc _ISR_SaveData_5
 MOVWR SAVEADDR5
 MOVFW STATUS
 MOVWR SAVEADDR5+1

 MOVFW op1
 MOVWR SAVEADDR5+2
 MOVFW op1+1
 MOVWR SAVEADDR5+3
 MOVFW op3+2
 MOVWR SAVEADDR5+4
 RET
.endproc

.proc _ISR_RestoreData_5
 MOVRW SAVEADDR5+2
 MOVWF op1
 MOVRW SAVEADDR5+3
 MOVWF op1+1
 MOVRW SAVEADDR5+4
 MOVWF op3+2

W_S:
  MOVRW SAVEADDR5+1
  MOVWF STATUS
  MOVRW SAVEADDR5
 RET

.endproc
</pre></td></tr>
</table>

- **F-Plane：**

<table>
<tr><th>有自動儲存功能</th><th>無自動儲存功能</th></tr>
<tr><td><pre>
.autoimport on
.export _ISR_SaveData_5,_ISR_RestoreData_5

#define _BANK_FLAG 5

 SAVEADDR5 = 0x7D ;__FPLANE_ADDRESS_MAX__-(__BANK1_OFFSET_VALUE__-0x80+4)

.fixcode +

.proc _ISR_SaveData_5

 BTFSC STATUS,_BANK_FLAG
 GOTO B1
</pre></td><td><pre>
.autoimport on
.export _ISR_SaveData_5,_ISR_RestoreData_5

#define _BANK_FLAG 5

 SAVEADDR5 = 0x7B

.fixcode +

.proc _ISR_SaveData_5

 BTFSC STATUS,_BANK_FLAG
 GOTO B1

 BSF STATUS,_BANK_FLAG
 MOVWF SAVEADDR5 ; save W
</pre></td></tr>
</table>

BSF STATUS,_BANK_FLAG MOVFW op1 MOVWF SAVEADDR5+0 MOVFW op1+1 MOVWF SAVEADDR5+1 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+2 BCF STATUS,_BANK_FLAG ret B1: MOVFW op1 MOVWF SAVEADDR5 MOVFW op1+1 MOVWF SAVEADDR5+1 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+2 RET .endproc .proc **_ISR_RestoreData_5** BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5 MOVWF op1 MOVFW SAVEADDR5+1 MOVWF op1+1 MOVFW SAVEADDR5+2 BCF STATUS,_BANK_FLAG MOVWF op3+2 RET .endproc .fixcode -	BCF STATUS,_BANK_FLAG MOVFW STATUS BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+1 MOVFW op1 MOVWF SAVEADDR5+2 MOVFW op1+1 MOVWF SAVEADDR5+3 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+4 BCF STATUS,_BANK_FLAG ret B1: MOVWF SAVEADDR5 MOVFW STATUS MOVWF SAVEADDR5+1 MOVFW op1 MOVWF SAVEADDR5+2 MOVFW op1+1 MOVWF SAVEADDR5+3 BCF STATUS,_BANK_FLAG MOVFW op3+2 BSF STATUS,_BANK_FLAG MOVWF SAVEADDR5+4 RET .endproc .proc **_ISR_RestoreData_5** BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5+2 MOVWF op1 MOVFW SAVEADDR5+3 MOVWF op1+1 MOVFW SAVEADDR5+4 BCF STATUS,_BANK_FLAG MOVWF op3+2 BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5+1 MOVWF STATUS BTFSS STATUS,_BANK_FLAG GOTO L0 MOVFW SAVEADDR5 RET L0: BSF STATUS,_BANK_FLAG MOVFW SAVEADDR5 BCF STATUS,_BANK_FLAG RET .endproc .fixcode -

4. 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。為了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：主題將包含巨集定義、包含的檔案、條件式編譯指令。

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

pragma 指令 (#pragma)

這個指令是用來指定編譯器的特定選項功能。這些選項是給特定的平臺和編譯器使用。如果＃pragma 的參數是編譯器不支援的，#pragma 指令將會被忽略並且不會產生任何錯誤或警告消息。

- #pragma tableromaddr

此 pragma 允許您設定全域常數變數的 TABLE ROM 開始位址，傳入 off 表示關閉該選項，如下所示：

```
聲明:          #pragma  tableromaddr  (const_variable_start_address)
               #pragma  tableromaddr  (off)
```

- #pragma tableromdt

此 prama 允許您將指標指向全域常數變數，且於 ROM 中使用 dt 的格式存取，若未開啟則是使用 retlw 的方式存取。 (0.5.7 版本後支援)

聲明:	#pragma tableromdt (on)

Const int value=0x3FFF;// 宣告一個全域常數變數,下方表格使用 dt 宣告及 retlw 全域常數變數的差別

dt	Retlw
000004: 3FFF dt $3fff	000005: 18FF RETLW 0xFF 000006: 183F RETLW 0x3F
優點: 占 1 個 ROM 的大小，有利於變數使用 缺點: dt 數值最多只到 3FFF，僅支援 int、unsigned int、short、unsigned short 4 種型態 Int、short 範圍:-8191~8191 unsigned int 、unsigned short 範圍:0~16383 並非支援全部 IC，僅支援特定 IC 使用	優點: 全域變數範圍與一般區域變數範圍相同 Char 範圍:-128~128 Unsigned char 範圍:0~255 Int、short 範圍:-32767~32767 Unaigned int、unsigned short 範圍:0~65534 Long 範圍:- 2147483647~2147483647 Unsigned long 範圍:0~4294967295 缺點: 如使用 int 型態時，會佔據 2 個 ROM 的大小 Long 型態會佔據 4 個 ROM 的大小 會佔據較多 ROM 空間

dt	Retlw
支持 tablerom IC: TM57ME16 TM57ME16AS TM57ME18 TM57MA25 TM57MA28 TM57MA28B TM57MA29 TM57MA29C TM57MA21B TM57MA15 TM57MA16 TM57MA1668 TM57MA1672 TM57M5526C TM57M5536C TM57MA17 TM57MA18 TM57P8620 TM57P8625 TM57P8640 TM57P8645 TM57M5620 TM57M5625 TM57M5640 TM57M5645 TM57M5610 TM57M5615 TM57ME15B TM57ME15CG TM57MA45 TM57MA46 TM57MA33 TM57M5541 TM57M5551	支持所有 IC

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，其無法藉由標準 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，來提高程式運行的效率。

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瞭解實際應用。

在 TM57 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，其回傳值將存在暫存器 op2。

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 Sum (int, int*);
main()
{
  int a=255,b=20,c=0;
  c= Sum (a,&b);
}
```

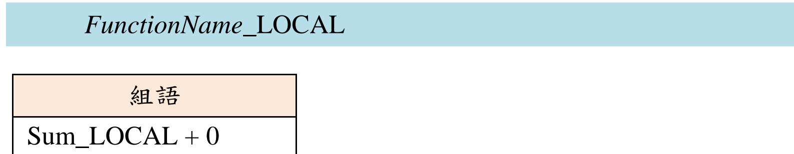
組語函式：

```
.autoimport    on
.debuginfo     on

.export        _Sum
.declfunc   Sum(2,3)
.CODE
.proc _Sum

        MOVFW Sum_PARAM+1
        MOVWF Sum_LOCAL
        MOVFW Sum _PARAM+2
        MOVWF Sum_LOCAL +1
        MOVFW Sum_PARAM+0
        MOVWF FSR
        MOVFW $00
        ADDWF Sum_LOCAL,1
        BTFSC STATUS,0
        INCF  Sum_LOCAL +1,1
        INCF  FSR,1
        MOVFW $00
        ADDWF Sum_LOCAL +1,1
        MOVFW Sum_LOCAL
        MOVWF op2
        MOVFW Sum_LOCAL +1
        MOVWF op2+1
        RET

.endproc
```

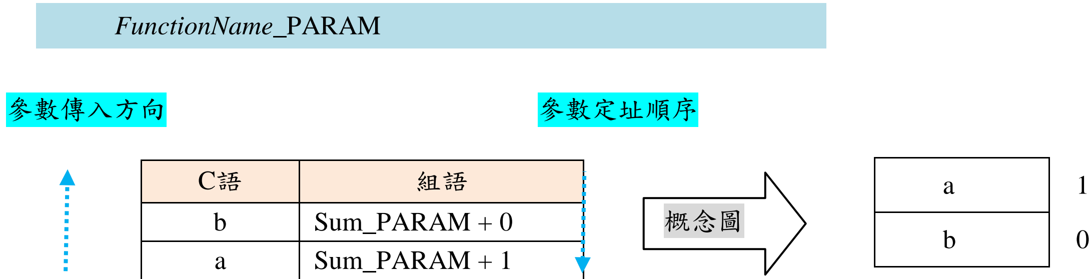
其中 .declfunc Sum (2,3)，第一個參數意指**區域變數**所佔之記憶體大小；第二個參數意指**傳入參數**大小（byte 為單位）。上例中，依傳入參數之各個資料型態，來計算記憶體大小

3 = 2 (int) + 1(char*)。區域變數來暫存所傳入之 int 數值，其記憶體大小為 2（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



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始位址如下表



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式的運行效率，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式以及程式具有在不同平臺間的可攜性（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。

TM57 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。

(三) 避免在 C/ASM 混合開發時, 使用.org xx 指令

組合語言的定址模式預設為重定址模式(reallocate address mode) 。因而，一般純組合語言程式會在程式的前面加入 .org xx 指令，以切換到絕對定址模式(absoute address mode) 。

例如

```
.org
      goto start
start:
 …..
```

但在 C/ASM 混合程式設計時，強烈建議不要使用 asm(“.org xx”) ，以避免不可預期的錯誤。

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 TM57 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：

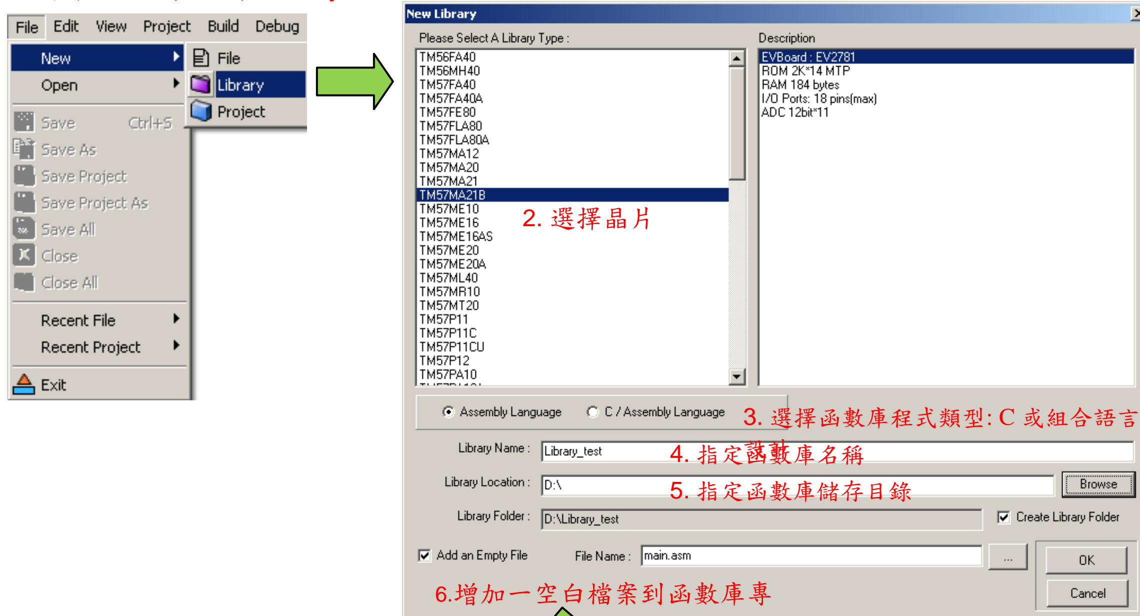
- 將相關運算模組函式集中在單一函式檔中，使得程式管理與維護更為簡單。
- 減少函式重複開發的時間，在函式文件說明上也更為有組織。
- 達到程式共用的目的，有效率的開發系統並縮短開發時程。

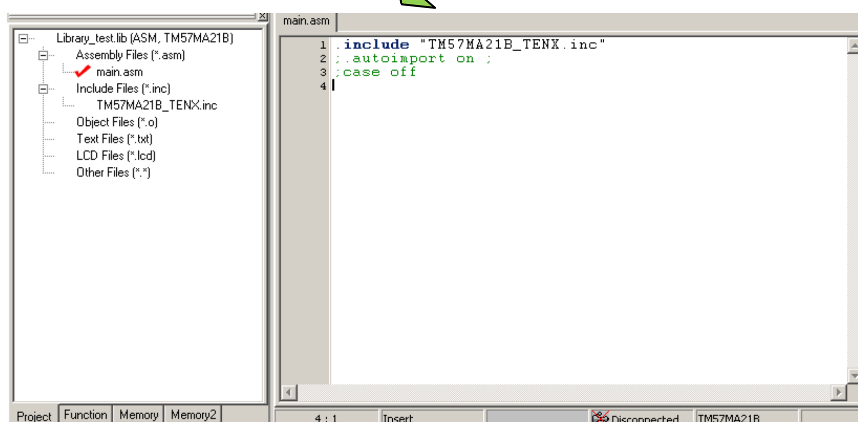
建立函式庫之方式

C 語言或組語函式皆可藉由 TICE99 IDE 所提供之工具以建立函式庫檔，有兩個建立函式庫的方法，其方法與概念如下圖：

方法一:新建立一個函式庫

1. 選擇 File|New|Library，在顯示的 New Library 視窗中輸入函數庫資訊，包含 ic 類型、C/組合語言、指定或創建函數庫目錄…等。
2. 建立函數庫之後，與一般專案相似，選擇性決定要加入至庫函數的 c/asm 檔案或是 object file (*.o)…等。
3. 完成編輯相關檔案後，即可編譯檔案以形成*.lib 檔。

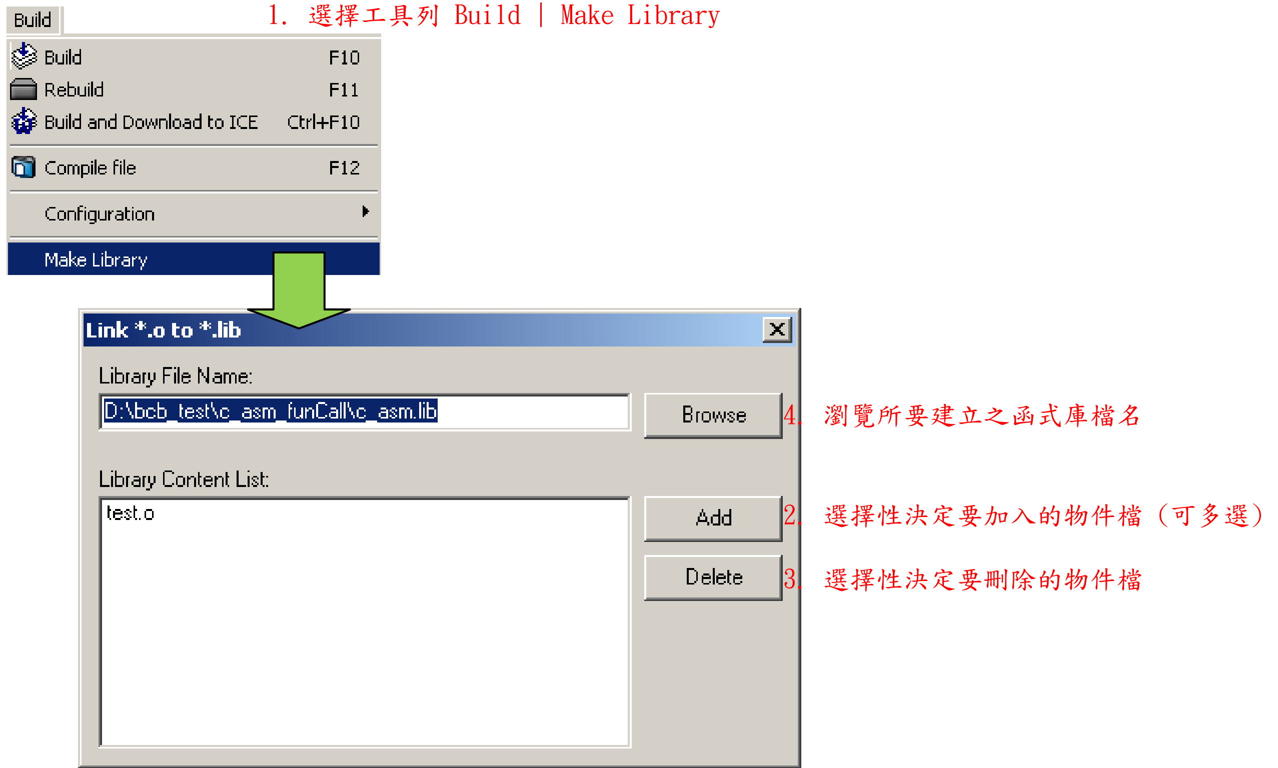




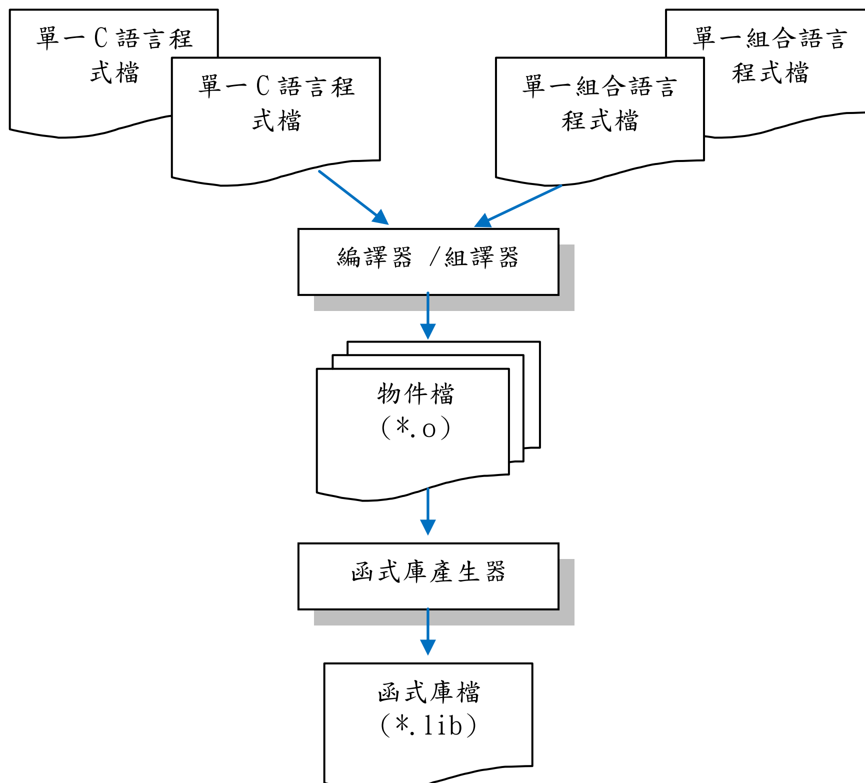
方法二：利用”函式庫產生器”工具來建立函式庫

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 TICE99 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）。



概念圖：



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：

1. 選取專案管理中的函式庫檔案項目
2. 按取右鍵，由彈出視窗中選取加入現存檔案
3. 選擇所要引入的函式庫檔
4. 完成選擇時，按取開啟鍵
5. 取消選擇時，按取取消鍵
6. 引入的函式庫檔名，出現在專案管理員中

5. 記憶體對應圖

在此章節中，僅圖示 C 編譯器在運算過程及因觸發中斷需要暫存及回存資料，而作用到的記憶體位址；各個單晶片之詳細記憶體對應圖，請參閱各晶片之使用手冊。

- 運算過程可能會使用到的暫存器位址：FSR、RSR、OP1~OP4、tmp1 及 stkptr。

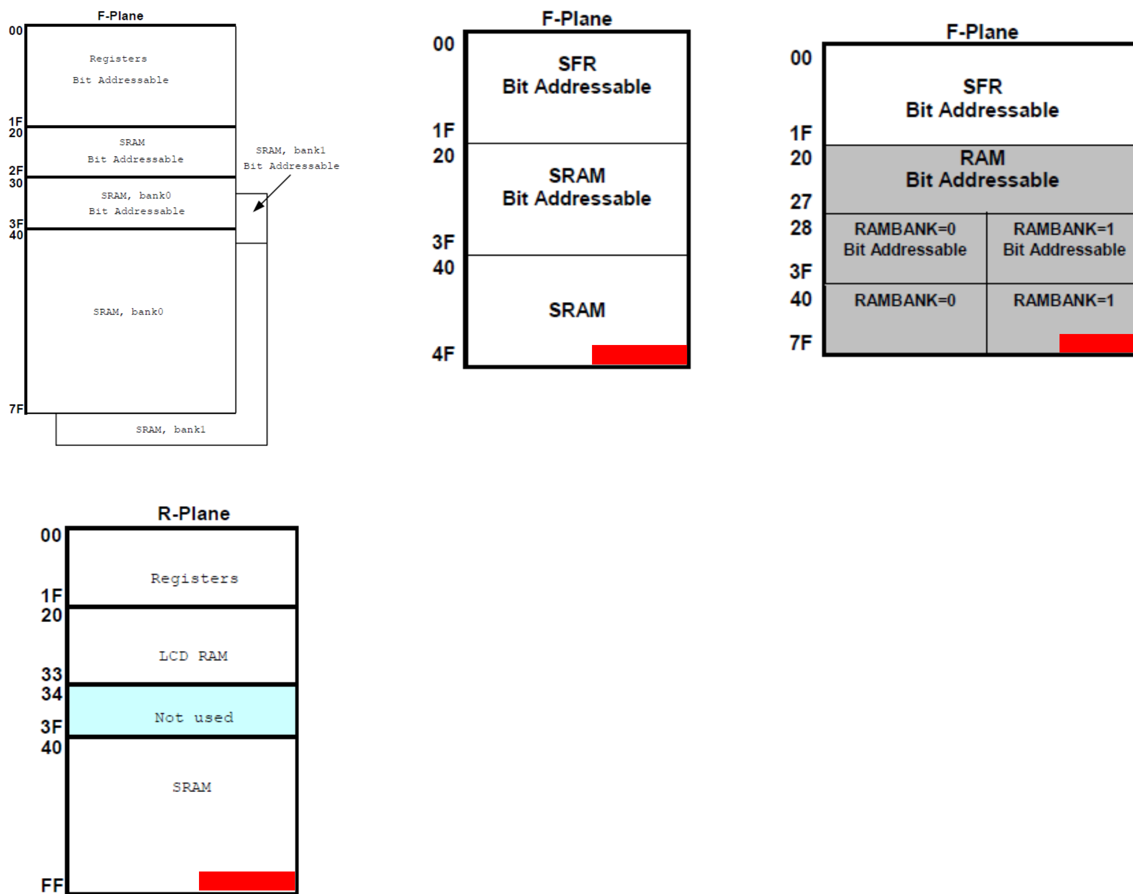
	0	1	2	3	4	5	6	7	8	9	a	b	C	d	e	f
0					FSR			RSR								
10																
20	OP1				OP2				OP3				OP4			
30	tmp1				stkptr (變動)											
…																

注意：

1. **當操作之 MCU 其 R-Plane 可利用 MOVWR 和 MOVRW 指令做記憶體存取時，才會儲存 RSR 暫存器之內容。**
2. **儲存 OP1~OP4、tmp1 及 stkptr 的寻址范围为变动的，其与程序运算复杂度有正对应的关系，详情请参阅 Fplane / Rplane 定义之说明。**

- 因觸發中斷需要暫存及回存資料，而使用到的暫存器位址（如紅色色塊所示）：

R-Plane	F-Plane Bank0	F-Plane Bank1



附錄

例子 1

- 計算字串長度

```
int strlen(char *tar)
{
        // Clear op2;
        asm("clrf op2");
        asm("clrf op2+1");

        asm("_strlen_LOOP:");
       // Read from source
        asm("movfw %o", tar);

      // Read value of tar address
        asm("call runtime_Ind_Read");
        asm("movwf op3");

       // Check end
```

```
        asm("testz op3");
        asm("btfsc STATUS, ZERO_FLAG");
        asm("ret");

        // Next
        asm("incf %o,1", tar);
        asm("incf op2,1");
        asm("goto _strlen_LOOP");
}
```

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

```
char *strcpy(char *tar,char *src)
{
      // Return tar value from op2 ( 0x24 )
       asm("movfw %o",tar);
      // return tar pointer in op2 address ( 0x24)
       asm("movwf op2");
       asm("_strcpy_LOOP:"); // generate label name
      // Read from source
       asm("movfw %o",src); // Set offset of LOCAL name src
       asm("call runtime_Ind_Read"); // call indirect read
       asm("movwf op3"); // op3 to write to target
      // Save to target
      // Set offset of LOCAL name tar ( strcpy_LOCAL+1 )
       asm("movfw %o",tar);
       asm("call runtime_Ind_Write"); // call indirect write
      // Check end
       asm("testz op3");
       asm("btfsc STATUS, ZERO_FLAG");
       asm("ret");
      // Next
       asm("incf %o,1",src);
       asm("incf %o,1",tar);
       asm("goto _strcpy_LOOP");
}
```

- 比較兩字串是否相同

```
int *strcmp(char *tar,char *src) // 比較兩字串是否相同
{
        asm("_strcmp_LOOP:"); // generate label name

       // Get tar value from op3 ( 0x28 )
        asm("movfw %o",tar);
        asm("call runtime_Ind_Read"); // call indirect read
```

```
        asm("movwf op1");

       // Set offset of LOCAL name src
        asm("movfw %o",src);
        asm("call runtime_Ind_Read");
        asm("testz op1");
        asm("btfsc STATUS,2");
        asm("ret");

        asm("subwf op1,0");
        asm("btfsc STATUS,2");
        asm("goto LZ1");
        asm("goto LZ0");

        // When  ZF is Zero
        asm("LZ0:");
        asm("movlw $FF");
        asm("btfsc STATUS,0");
        asm("xorlw $FE");
        asm("movwf op2");
        asm("btfss op2,7");
        asm("movlw $00");
        asm("movwf op2+1");
        asm("ret");

        // When  ZF is one
        asm("LZ1:");
        asm("btfsc STATUS,0");
        asm("movlw $01");
        asm("xorlw $01");
        asm("movwf op2");
        asm("clrf  op2+1");
        asm("incf %o,1",src);
        asm("incf %o,1",tar);
        asm("movfw %o",tar);
        asm("call runtime_Ind_Read");
        asm("movwf op1");
        asm("testz op1");
        asm("btfsc STATUS,2");
        asm("ret");
        asm("goto _strcmp_LOOP");
}
```

- 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 tar 所指之字串）之後。

```
// 把src字串放在tar字串後
char *strcat(char *tar,char *src)
{
   // Return tar value from op2 (0x24)
   asm("movfw %o",tar);
   asm("movwf op2");

   // Check the char of tar string is '\0'
   asm("START:");
   asm("movfw %o",tar);
   asm("call runtime_Ind_Read");
   asm("movwf op1");
   asm("testz op1");
   asm("btfss STATUS,2");
   asm("goto tarnext_LOOP");

   asm("_strcat_LOOP:");
   asm("movfw %o", src);          // Set offset of LOCAL name src
   asm("call runtime_Ind_Read");  // call indirect read
   asm("movwf op3");              // op3 to write to target

   asm("testz op3");
   asm("btfsc STATUS, ZERO_FLAG");
   asm("goto add_null");
   asm("movfw %o", tar);
   asm("call runtime_Ind_Write");  // call indirect write
   asm("testz op3");
   asm("goto Next_LOOP");

   asm("Next_LOOP:");
   asm("incf %o", tar);
   asm("incf %o", src);
   asm("goto _strcat_LOOP");

   asm("tarnext_LOOP:");
   asm("incf %o", tar);
   asm("goto START");

   asm("add_null:");
   asm("movlw $00");
   asm("movwf op3");
   asm("movfw %o", tar);
   asm("call runtime_Ind_Write");
   asm("ret");
}
```

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 WRKeyData1Bytes。

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。

```
main()
{
    WRKeyData1Bytes();
}
```

組語函式：

```
;WRKeyData1Bytes
        .autoimport    on
        .debuginfo     on
        .export        _WRKeyData1Bytes ; leader char is  _ ( at prefix )
        ; use ".declfunc" directive to allocate the size of local and parameter.
        ; format: .declfunc funname(local_size,param_size)
        .declfunc      WRKeyData1Bytes(0,0)

.CODE
.proc _WRKeyData1Bytes
        movlw     40h
        addwf     79h,0
        movwf     FSR
        bsf       03h,5   ;STATUS,RAMBANK
        movfw     7Fh
        movwf     00h  ;INDF
        bcf       03h,5  ;STATUS,RAMBANK
        movlw     .1
        addwf     79h,1
        ret
.endproc
```

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（BYTE 為單位），宣告語句為 .declfunc strcpy(0,2)。

```
; 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(2) = 2
        ; declared 2 bytes space to parameter
        .declfunc       strcpy(0,2)

.CODE

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

        movfw strcpy_PARAM+1 ; target
        movwf op2 ;  return tar pointer in op2 address (0x24)

LOOP:
       ;*** Read from source
        movfw strcpy_PARAM+0  ; Set offset of LOCAL name src
        call runtime_Ind_Read ; call indirect read (call runtime library function)
        movwf op3             ; op3 to write to target

       ;*** Save to target
        movfw strcpy_PARAM+1   ; Set offset of LOCAL name tar
        call runtime_Ind_Write ;  call indirect write (call runtime library function)

       ;*** Check end ?
        testz op3
        btfsc STATUS, ZERO_FLAG
        ret
```

```
    ;*** Next
    incf strcpy_PARAM+0,1
    incf strcpy_PARAM+1,1
    goto LOOP

.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+1 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 = 40h
 tar_str = 45h

 movlw 'A'
 movwf src_str
 movlw 'B'
 movwf src_str+1
 movlw 'C'
 movwf src_str+2
 movlw 'D'
 movwf src_str+3
 movlw 0
 movwf src_str+4

 ;*********************************************
 ;*** pass parameter from right to left
 ;*** pointer size is 1 byte
 ;*********************************************
 movlw src_str
 movwf strcpy_PARAM+0 ; store source address to PARAM+0
 movlw tar_str
 movwf strcpy_PARAM+1 ; store target address to PARAM+1
 call  _strcpy

loop:
  nop
  goto loop
```