



MICROCHIP

MPLAB-C18 Workshop

第二天課程

中斷程式的處理

MPLIB-C18 函數庫

用 **C** 來完成一個應用程式

版本 : W401 Rev. 3

日期 : Oct. 25 2010

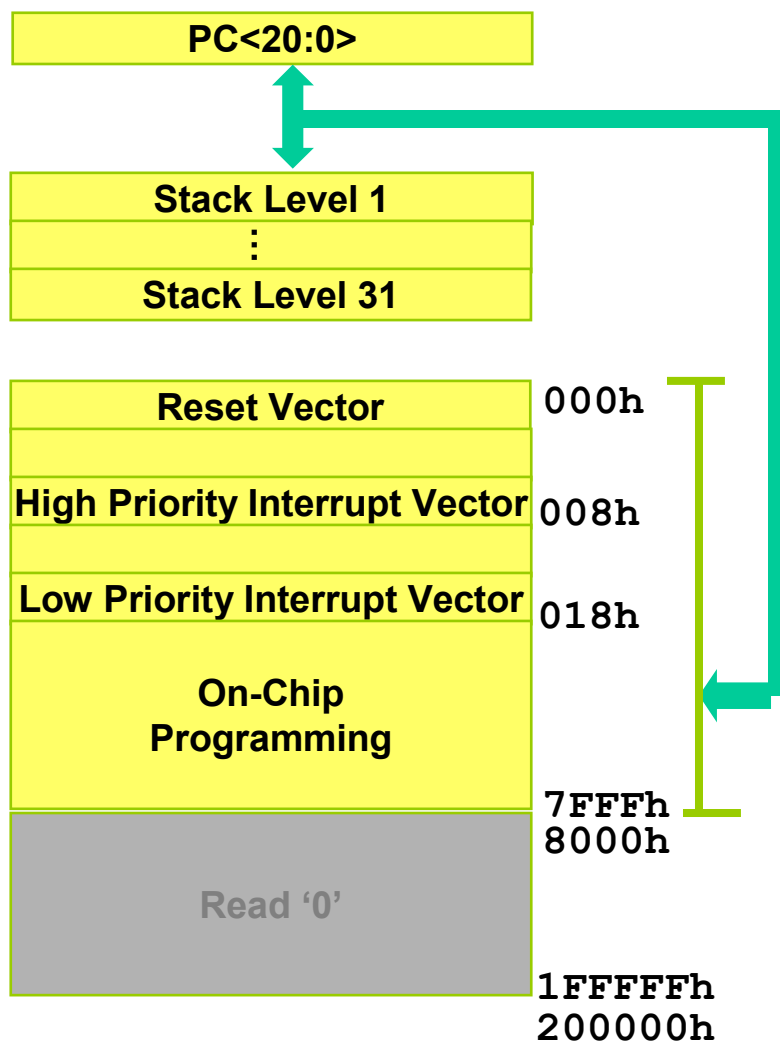
第五章

中斷程式的處理

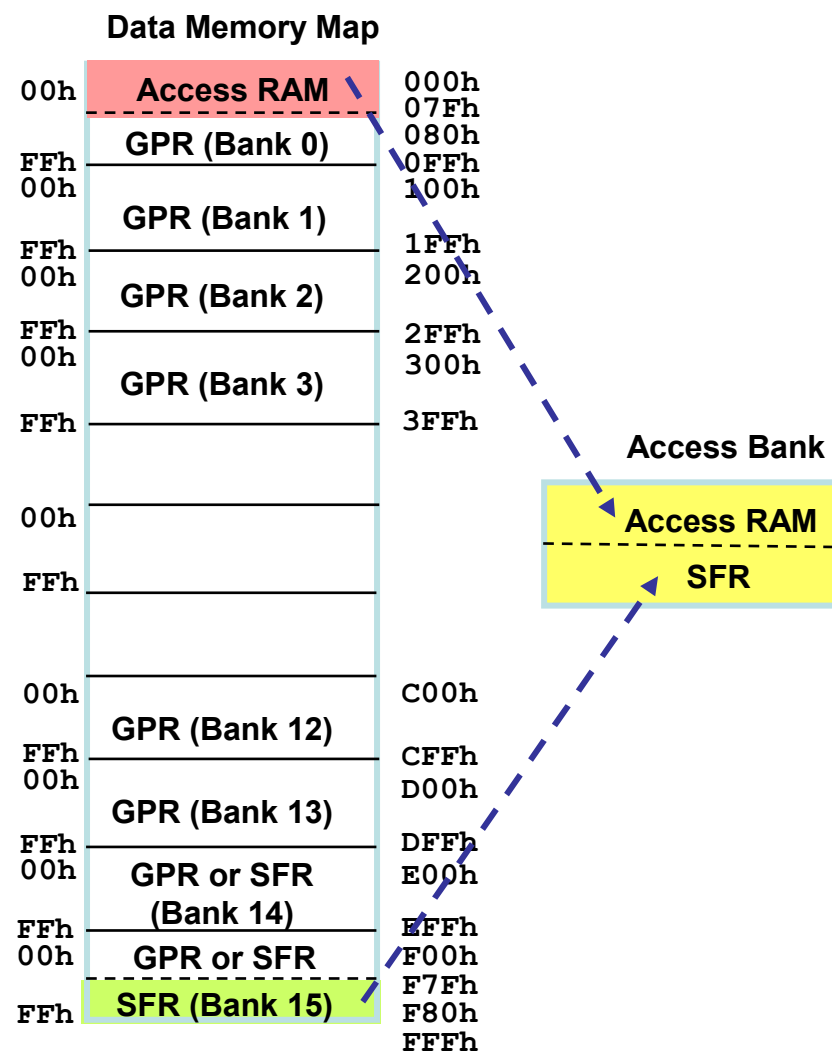
1. 安排變數資料
2. 安排常數資料
3. 安排程式位置
4. 處理中斷程式
5. 串列通訊應用

18C452記憶體的配置

程式記憶區



資料記憶區



安排變數的位置 (**RAM Location**)

- 在**Access Bank**內的變數執行速度較快
- 利用前置處理指令“**#pragma {udata/idata}**”來指定變數在**RAM**的特定位置

```
#pragma udata [data-qualifier] [section-name [=address]]
```

```
#pragma idata [data-qualifier] [section-name [=address]]
```

- 利用前置處理指令“**#pragma udata/idata**”來結束指定節區位置的作業

```
#pragma udata/idata
```

#pragma udata

- #pragma udata [data-qualifier] [section-name [= addr]]
 - **#pragma udata [xxx]** 會將以下所宣告的變數作特定的位置安排，直到遇到指令“#pragma udata”才進行常態安排
 - **udata** : 設定無初始值的變數(idata有初始值的變數)

- Option*
- **[data-qualifier]** : 變數要擺在那一區域
 - “access” ==> 安排在 ACCESS Bank 的 RAM (0x00-0x7F) 中
 - “空白” ==> 安排在非 ACCESS Bank (GPR)

- Option*
- **[section-name [= addr]]** : 變數放置的位址
 - 指定section-name: 與 Linker的連結描述檔內的section-name相同，則會依連結描述檔內的指定進行安排
 - 不指定section-name: 即獨立的名字，並不與 Linker的連結描述檔內的section-name相同，此時變數會放在 unprotected區
 - = addr : 在不指定section-name時，可以強制設定變數位址

#pragma udata 宣告（範例）

```
#pragma udata access AccessSection  
near unsigned char Temp_Code[4];  
near unsigned char Rec_Data;  
near unsigned char PWM_Duty;  
near unsigned char On_Flag;
```

宣告以下之變數放在Access Bank
中，由Linker自行安排位址

```
#pragma udata abc=0x100  
unsigned char j;  
unsigned char i;  
unsigned char e;  
unsigned char f;
```

宣告以下之變數放在GPR中
位址為0x100的地方

```
#pragma udata test  
unsigned char EE_Write_Data;  
unsigned char EE_Addr;  
unsigned char Send_UR;  
unsigned char Err;
```

宣告以下之變數放在GPR中，由Linker自行
安排位址，又 section name 有特別指定故會
被安排在 Bank 2 的位址

SECTION NAME=test RAM=gpr2

```
#pragma udata
```

變數實際位址（範例）

Name	Address	Location	Storage File
Temp_Code	0x000000	data	extern D:\WORKSH~1\402\TEST\TEST1.C
Rec_Data	0x000004	data	extern D:\WORKSH~1\402\TEST\TEST1.C
PWM_Duty	0x000005	data	extern D:\WORKSH~1\402\TEST\TEST1.C
On_Flag	0x000006	data	extern D:\WORKSH~1\402\TEST\TEST1.C
USART_Status	0x000093	data	extern ..\pmc\USART\18Cxx\USARTD.C
j	0x000100	data	extern D:\WORKSH~1\402\TEST\TEST1.C
i	0x000101	data	extern D:\WORKSH~1\402\TEST\TEST1.C
e	0x000102	data	extern D:\WORKSH~1\402\TEST\TEST1.C
f	0x000103	data	extern D:\WORKSH~1\402\TEST\TEST1.C
EE_Write_Data	0x000200	data	extern D:\WORKSH~1\402\TEST\TEST1.C
EE_Addr	0x000201	data	extern D:\WORKSH~1\402\TEST\TEST1.C
Send_UR	0x000202	data	extern D:\WORKSH~1\402\TEST\TEST1.C
Err	0x000203	data	extern D:\WORKSH~1\402\TEST\TEST1.C
PORTAbits	0x000f80	data	extern C:\MCC18\SRC\PROC\p18c452.asm

安排常數資料的位置 (rom-data Location)

- 利用前置處理指令“**#pragma romdata**”來指定常數資料在**ROM**的位置

#pragma romdata [section-name [=address]]

- 一般常用的固定不變資料，例：查表資料、顯示的字串資料、陣列常數...等

- 利用前置處理指令“**#pragma romdata**”來結束此一指定位置作業

#pragma romdata

#pragma romdata (範例)

```
#pragma romdata RomDataSpace=0x400    // 設定 romdata 起始位址在 0x400
rom unsigned char Array1[20]= {0x0F,0x0E,0x0D,0x0C,0x0B,0x0A,0x09,0x08,
                                0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};

#pragma romdata                          // 恢復為一般 romdata section

unsigned char Count;
unsigned char Array2[20];                // 宣告陣列變數

void main(void)
{
    Count = 0x00;
    while(Array1[Count++])                // 檢查字元 =0x00 ?
    {
        Array2[Count] = Array1[Count]; // 將 ROM 陣列資料存入 RAM 陣列中
    }
}
```

#pragma romdata

(範例-實際位址)

Section Info					
Section	Type	Address	Location	Size(Bytes)	
-----	-----	-----	-----	-----	
.cinit	romdata	0x00002a	program	0x000002	
.code_PRAGMA.o	code	0x000038	program	0x00006e	
.romdata_PRAGMA.o	romdata	0x0000a6	program	0x000002	
.idata_PRAGMA.o_i	romdata	0x0000a8	program	0x000000	
RomDataSpace	romdata	0x000400	program	0x000010	
.tmpdata	udata	0x000000	data	0x000002	
.udata_PRAGMA.o	udata	0x000080	data	0x000000	
.idata_PRAGMA.o	idata	0x000080	data	0x000000	
.stack	udata	0x000500	data	0x000100	
SFR_UNBANKED0	udata	0x000f80	data	0x000023	
SFR_UNBANKED1	udata	0x000fab	data	0x000055	

安排程式的位置 (code Location)

- 利用前置處理指令“**#pragma code**”來指定程式在**ROM**的位置

#pragma code [section-name [=address]]

- **section-name**可以在連結描述檔中加以指定該段程式編譯後的執行位址 (**18c452.1kr**)
- 也可以直接指定該段程式的執行位址

● **#pragma code My_Code_On = 0x1000**

- 利用前置處理指令“**#pragma code**”來結束此一指定位置作業

#pragma code

設定 EEPROM 燒錄值

- 在 **lkr** 檔裡有對 **Configuration Bits, Device ID** 及內建 **EEPROM** 的特殊位址宣告 (**Code Page**)

CODEPAGE	NAME= config	START=0x300000	END=0x30000D	PROTECTED
CODEPAGE	NAME= devid	START=0x3FFFFE	END=0x3FFFFFF	PROTECTED
CODEPAGE	NAME= eedata	START=0xF00000	END=0xF000FF	PROTECTED

- 所以設定 **EEPROM** 初始直直程式裡 (底下的 **EEPROM** 設定值會與程式碼一起燒入 **PIC18F4520** 裡)

```
#pragma romdata eedata
```

```
rom unsigned char EE_Dara1[ ] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};
```

```
rom unsigned char EE_Data2[ ] = {0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
```

```
rom unsigned char EE_Data3[ ] = {'H','i','-','T','e','c','h',0x00,};
```

```
rom unsigned int EE_Data4 = 0xAAA0,EEdata = 0xAAA2;
```

```
rom unsigned long EE_Data6 = 0x61626364;
```

```
#pragma romdata
```

Configuration Bits 的設定

- **Config. 的位址定義在 PIC18F4520.lkr**

- CODEPAGE NAME=config START=0x300000 END=0x30000D PROTECTED
- SECTION NAME=CONFIG ROM=config

- 設定 Configuration Bits 請參考 MPLAB IDE 裡的
Help → Topics → Language Tools
→ PIC18F config Setting 裡的說明。

使用範例:

#pragma config	OSC=HS
#pragma config	PWRT=OFF
#pragma config	BOREN=OFF
#pragma config	WDT=OFF
:	
:	

Configuration Bits 設定項

hlpPIC18ConfigSet

隱藏頁籤 上一頁 下一頁 首頁 列印

內容(C) 索引(N) 搜尋

- ? PIC18F44J11
- ? PIC18F44J50
- ? PIC18F44K20
- ? PIC18F4510
- ? PIC18F4515
- ? PIC18F452
- ? PIC18F4520
- ? PIC18F4523
- ? PIC18F4525
- ? PIC18F4539
- ? PIC18F4550
- ? PIC18F4553
- ? PIC18F458
- ? PIC18F4580
- ? PIC18F4585
- ? PIC18F45J10
- ? PIC18F45J11
- ? PIC18F45J50
- ? PIC18F45K20
- ? PIC18F4610
- ? PIC18F4620
- ? PIC18F4680
- ? PIC18F4682
- ? PIC18F4685
- ? PIC18F46J11
- ? PIC18F46J50

PIC18F4520

Oscillator Selection bits:

OSC = LP	LP oscillator
OSC = XT	XT oscillator
OSC = HS	HS oscillator
OSC = RC	External RC oscillator, CLKO function on RA6
OSC = EC	EC oscillator, CLKO function on RA6
OSC = ECI06	EC oscillator, port function on RA6
OSC = HSPLL	HS oscillator, PLL enabled (Clock Frequency = 4 x FOSC1)
OSC = RCI06	External RC oscillator, port function on RA6
OSC = INTI067	Internal oscillator block, port function on RA6 and RA7
OSC = INTI07	Internal oscillator block, CLKO function on RA6, port function on RA7

Fail-Safe Clock Monitor Enable bit:

FCMEN = OFF	Fail-Safe Clock Monitor disabled
FCMEN = ON	Fail-Safe Clock Monitor enabled

練習 5-1

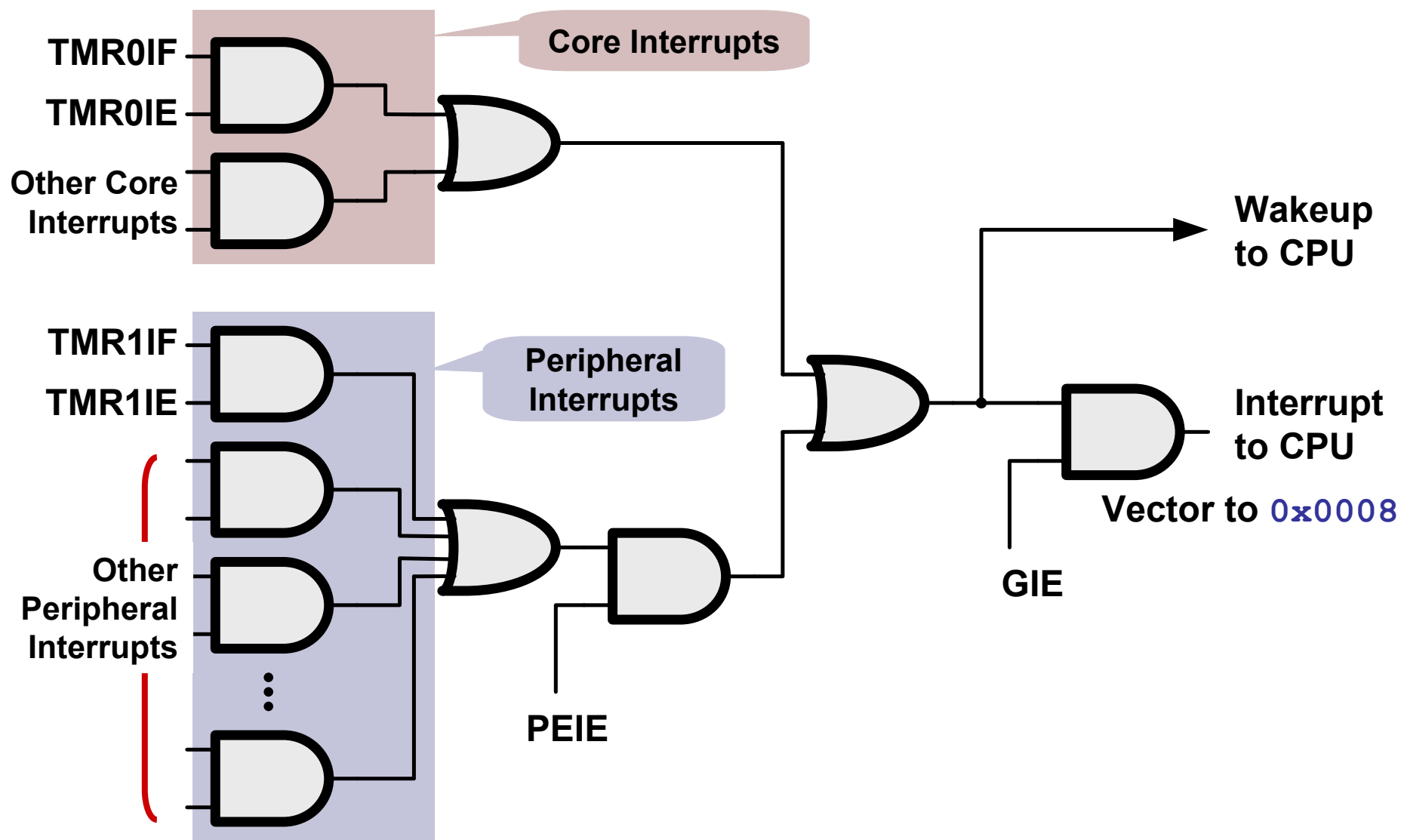
- 開啓 “\RTC\W401\Ans5-1\Ex5-1.mcp”
- 利用“**#pragma**”指令將變數及程式編譯到特定的位址：
 - LCD_MSG1 放在ROM中，位址為0x1000
 - LCD_MSG2 字串變數放在RAM中，位址為0x100
 - 其餘變數 AD_Temp、LCD_Temp、AD_Result 則放在 Access Bank中
 - 程式中的函數從位址0x2000開始編譯
- 檢查編譯結果，並測試執行結果是否正常
 - Ex5-1.map & Ex5-1.lst
 - 找不到map檔如何處置？

18F4520 中斷處理

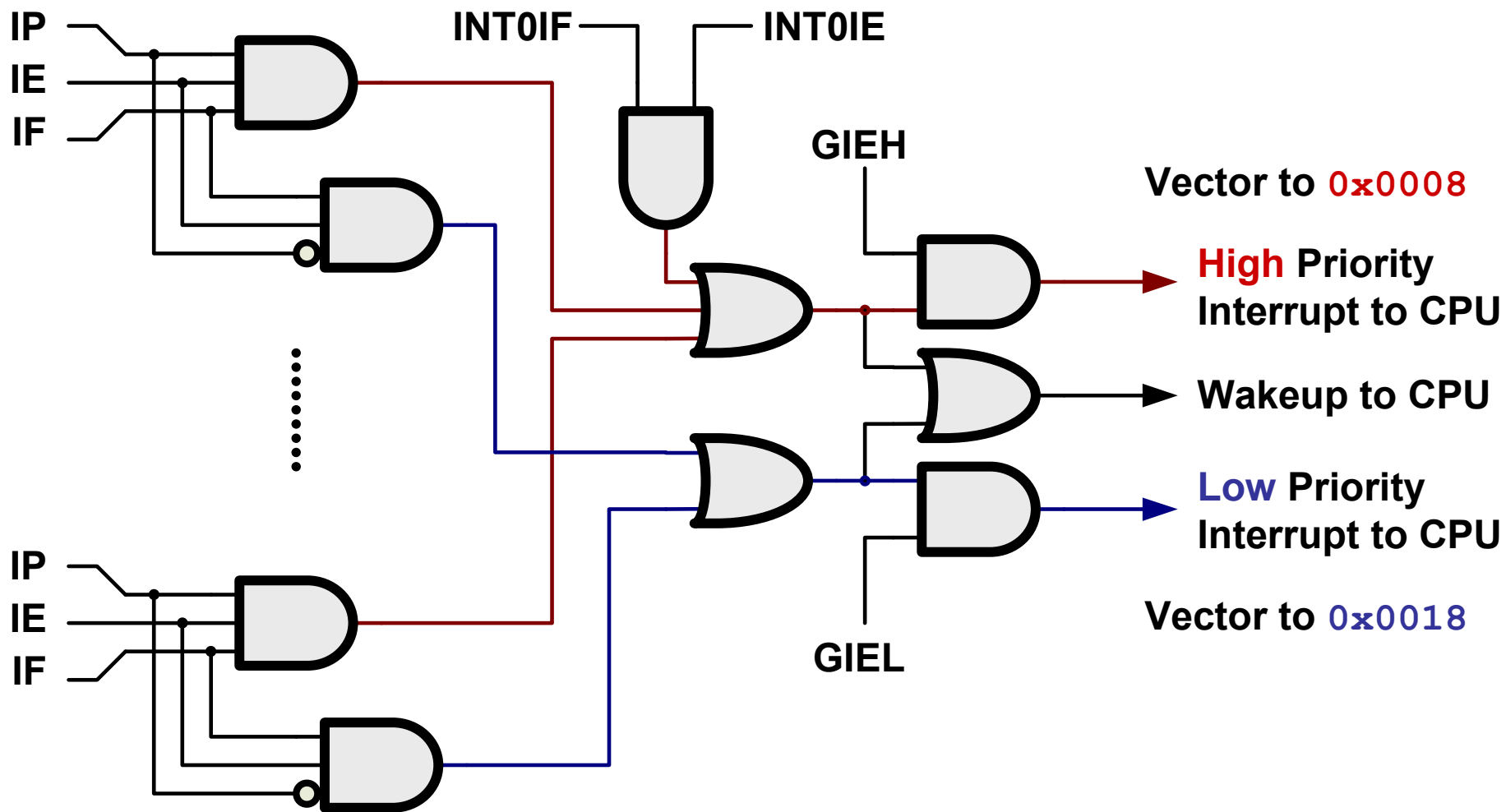
- **18F4520**有兩個中斷向量點
 - 高優先權==>中斷向量位址0x0008
 - 低優先權==>中斷向量位址0x0018
 - 每個中斷源均可選擇其中斷優先權（二選一）
 - 每個中斷源均有獨立的中斷旗標(Flag)
 - 中斷旗標的清除==>自行用軟體清除
 - 每個中斷源均可Enable或Disable
- 當然**PIC18**系列也可設定與**PIC16Fxxx**系列的中斷相容（關掉優先權的設定）

中斷邏輯電路

傳統模式



中斷邏輯電路 優先權模式



Shadow 暫存器

- **18F4520**有“**Shadow Register**”的設計，能提高中斷程式對事件的反應速度
- 高優先權中斷
 - W，BSR，STATUS 的存入/取出使用Shadow Register
 - 程式的返回：retfie FAST
- 低優先權中斷
 - W，BSR，STATUS 的存入/取出則必需透過軟體堆疊
 - 程式的返回：retfie 0
- 以上處理，在 **C18** 編譯器均打點完畢，程式設計者無需傷神考慮
- 結論：**C** 的中斷處理比組合語言更簡單方便

設定高優先權中斷服務程式

- 利用前置處理指令“**#pragma code**”來設定高優先權中斷向量位址=**0x0008**，並將控制權轉移給中斷服務程式
- 利用前置處理指令“**#pragma interrupt**”來設定該函數為高優先權中斷服務程式(可在程式任何位址)，處理完畢會自行用**retfie FAST**回家

#pragma interrupt func-name save=symbol list

- **func-name** : 高優先權中斷服務程式名稱
- **save= symbol list** : 在中斷服務程式中，須被保存的變數(例: **save= FSR0, PRODL**)

高優先中斷設定

```
#pragma code hi_vector=0x0008    // 設定中斷進入點
```

```
void isr_high_code(void)
```

```
{  
    _asm  
    goto isr_high  
    _endasm  
}
```

使用內建組合語言功能，轉移控制權到中斷服務程式(isr_high)

```
#pragma code
```

```
/**/
```

```
/* Function: isr_high(void) */
```

```
/* - Received a serial data from RS-232 */
```

```
/* - Save the received data to Rec_Data */
```

```
/**/
```

```
#pragma interrupt isr_high
```

```
void isr_high(void)
```

```
{  
    Rec_Data=ReadUSART();  
    PORTD=Rec_Data;  
}
```

```
#pragma code
```

中斷服務程式(isr_high)

中斷的前置設定

- 由上一頁來看中斷服務的設定是很簡單
- 但別忘了還有一些暫存器須設定後，中斷才會真正的動作(以**USART**的接收為例)
 1. 開啓中斷優先權的設定: **RCONbits.IPEN=1;**
 2. 設定**USART**的接收爲高優先權: **IPR1bits.RCIP=1;**
 3. **enable USART**的接收中斷: **PIE1bits.RCIE=1;**
 4. 啓動高優先權中斷: **INTCONbits.GIEH=1;**
- 完成中斷的基本設定

設定低優先權中斷服務程式

- 利用“**#pragma code**”來設定低優先權中斷向量位址 **=0x0018**，並將控制權轉移給低優先權中斷服務程式
- 利用“**#pragma interruptlow**”來設定該函數為低優先權中斷服務程式，返回方式是 **retfie**

#pragma interruptlow func-name save=symbol list

- **func-name**：低優先權中斷服務程式名稱
- **save= symbol list**：在中斷服務程式中，須被保存的變數
(例: save= FSR0, PRODL)

中斷注意事項

- 中斷函數無法傳遞參數
- 中斷所使用的變數需加入 **volatile** 的宣告
- 中斷函數儘量不要使用 **Local** 變數，引響中斷響應時間
- 中斷越短越好，不要做太多的處理，可以設定 **Flag** 後交給主程式處理
- 中斷函數最好不要做加、減、乘、除的運算，因為這些運算會使用到 **tmpdata** 這就有可能與主程式相衝這塊 **Memory**

tmpdata vs 中斷

- 共享 tmpdata 在主程式與中斷函數
- 新版的 C18 會自動儲存 tmpdata，可防止中斷程式出錯，但也因要儲存 tmpdata 所以響應速度會變慢
- 如果中斷程式只是很簡單的處理並未使用到 tmpdata 就可以用底下的方式以加快中斷響應速度

```
#pragma interrupt_isr nosave=section(".tmpdata")  
void isr (void)  
{  
    ...  
}
```

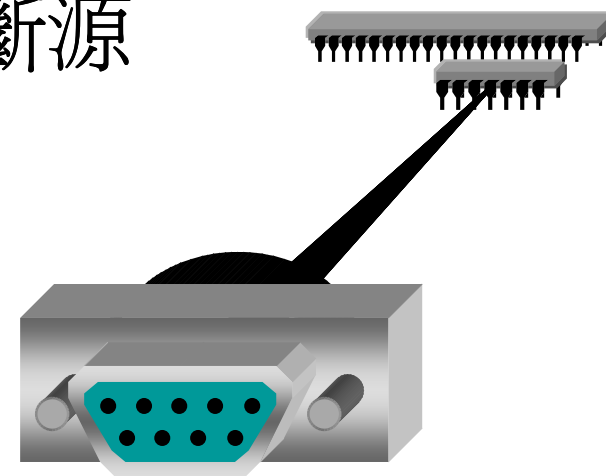
練習 5-2

- 開啓 “\RTC\W401\Ans5-2\Ex5-2.mcp”
- 將**Timer2**設定為中斷模式
 - 每8mS 中斷一次
- **LED**間隔**96mS**向左移**7bit**後再向右移**7bit**
- **LED**重複左、右移動(跑馬燈)

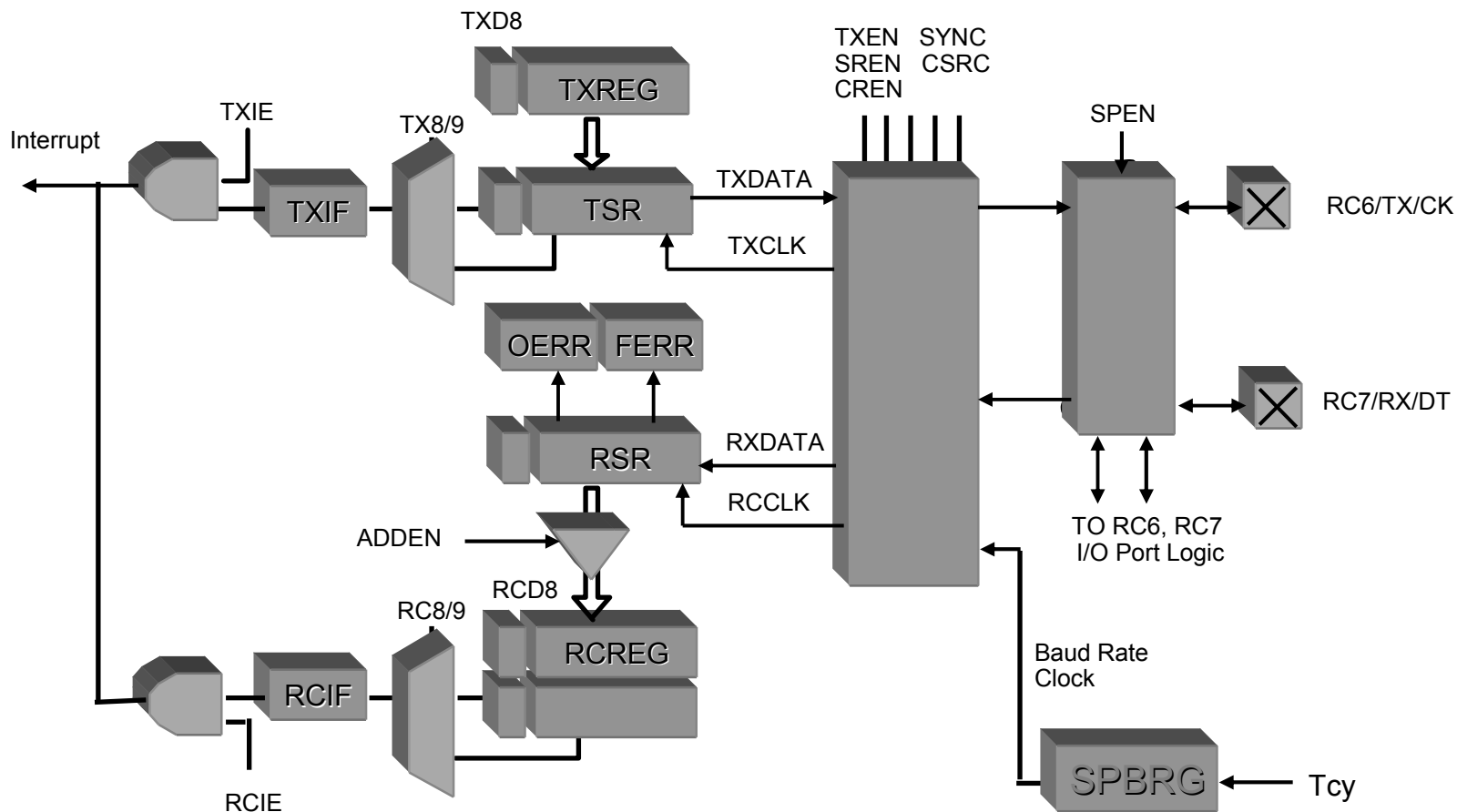
注意! 變數設定後並沒有在 **main()**函數內使用，
程式經 **optimize** 後變數會不見，要如何處理？

串列通訊功能

- 全雙工非同步串列或半雙工同步串列傳輸
- 串列接收與發送採用雙資料緩衝器的設計
- 串列接收與發送採獨立得中斷源
- **8-bit or 9-bit** 資料格式
- 獨立的鮑率產生器
- 最高速率 **@ 40MHz**
 - 同步傳送: 10M baud
 - 非同步傳送: 低速模式625K baud或高速模式2.5M baud
- 支援 **9-bit** 位址或資料的判斷模式

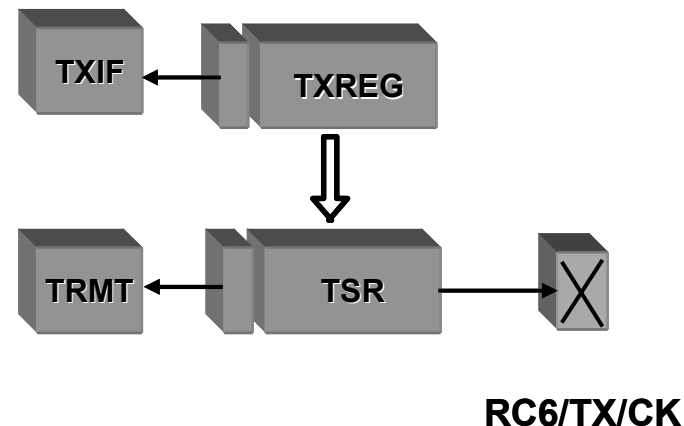


串列通訊方塊圖



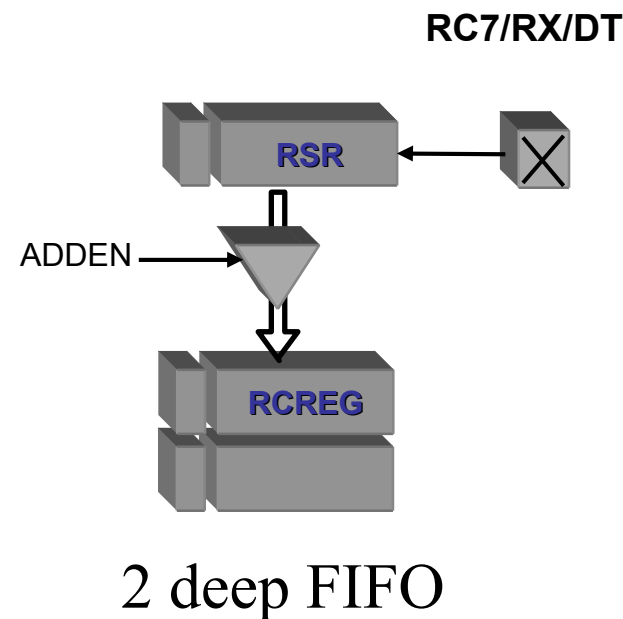
TXIF & TRMT 的動作

- 假如**TXREG**的資料被載到**TSR**，**TXREG**會空出;則**TXIF = 1**
- 假如**TSR**的資料串列傳送完畢;則**TRMT = 1**
- 假設**TXREG**剛載入資料時**TRMT**為空的(**TRMT=1**)，則這筆資料會立即被送到**TSR**，串列傳送會動作，同時**TXIF = 1**
- **TXIF**是可單獨使用，即使**USART**的**TX**中斷是關閉的(**TXIE=0**)
- 由以上動作可知偵測發送狀態**TXIF**會比**TRMT**來的快



RCIF的動作

- **RSR** 為一個移位器，接收**8-bits**的串列資料及 **start & stop bit**.
- 接收到的串列資料會被載入到 **RCREG FIFO** 並將設定 **RCIF**
- 假設上一筆資料在**FIFO**中未被拿走此時又有一筆串列資料接收近來，則這筆新的資料會被存在第二級的**FIFO**中
- 若兩級**FIFO**均有資料，接收中斷產生將第一級資料提走，中斷處理完畢後，第二級**FIFO**的資料會立即產生中斷



常用的 **USART** 函數庫

- **USART**的原始程式放在：
 - C:\MCC18\src\pmc_common\USART
- **OpenUSART**：開啓並設定**USART**的工作模式
- **BusyUSART**：測試發送是否忙線中？
- **putsUSART**：從**RAM**中提取字串並發送出去
- **putrsUSART**：從**ROM**中提取字串並發送出去
- **ReadUSART**：讀取接收的資料(Byte)
- **WriteUSART**：傳送一個資料 (Byte)

使用 USART 的函數庫(一)

● OpenUSART : 開啓並設定USART的工作模式

void OpenUSART (unsigned char config , char SPBRG)

➤ config中的定義字在usart.h檔案中定義

- USART_TX_INT_OFF ==> PIE1bits.TXIE=0;
- USART_RX_INT_ON ==> PIE1bits.RCIE=1;
- USART_ASYNC_MODE ==> TXSTAbits.SYNC=0;
- USART_EIGHT_BIT ==> TXSTAbits.TX9=0;
RCSTAbits.RX9=0;
- USART_CONT_RX ==> RCSTAbits.CREN=1;
- USART_BRGH_HIGH ==> TXSTAbits.BRGH=1;

➤ 以下兩個位元也會被設定以啓動USART

- RCSTAbits.SPEN=1; TXSTAbits.TXEN=1;

➤ 中斷優先權控制:

- RCONbits.IPEN=1; IPR1bits.RCIP=1;

➤ SPBRG = 103 : 速率產生器設爲9600 bps

- $f_{osc} / [(SPBRG+1)*16] = 16\text{MHz} / [(103+1)*16] = 9615\text{bps} (+0.16\%)$

使用 USART 的函數庫(二)

- **char ReadUSART (void): 讀取接收的資料(Byte)**

- ReadUSART 會自RCREG暫存器中讀取接收資料

例: `Rec_Buff = ReadUSART();`

- **void WriteUSART (char data): 傳送一個資料 (Byte)**

- 將 data 寫入到TXREG暫存器，並開始傳送資料

例: `WriteUSART('A')` ==> 發送 A 字元

`WriteUSART('\n')` ==> 發送 0x0A (換行字元)

- **char BusyUSART(void) : 測試發送是否忙線中?**

例: `while (BusyUSART()); // TXREG busy?`

`writeUSART('A')` // 發送 A 字元

使用 **USART** 的函數庫(三)

- **putsUSART** : 從**RAM**中提取字串並發送出去

`void putsUSART (char *dptr)`

- 傳入該函數的參數必須是一個指向RAM的指標 (Pointer)，即字串或陣列的起始位址
- 傳送字串直到 0x00 (null character)

- **putrsUSART** : 從**ROM**中提取字串並發送出去

`void putrsUSART (const rom char *dptr)`

- 傳入該函數的參數必須是一個ROM的指標 (Pointer)

例: `const rom char Msg0[]="Hello Word!";`
`putrsUSART(Msg0);`

使用超級終端機

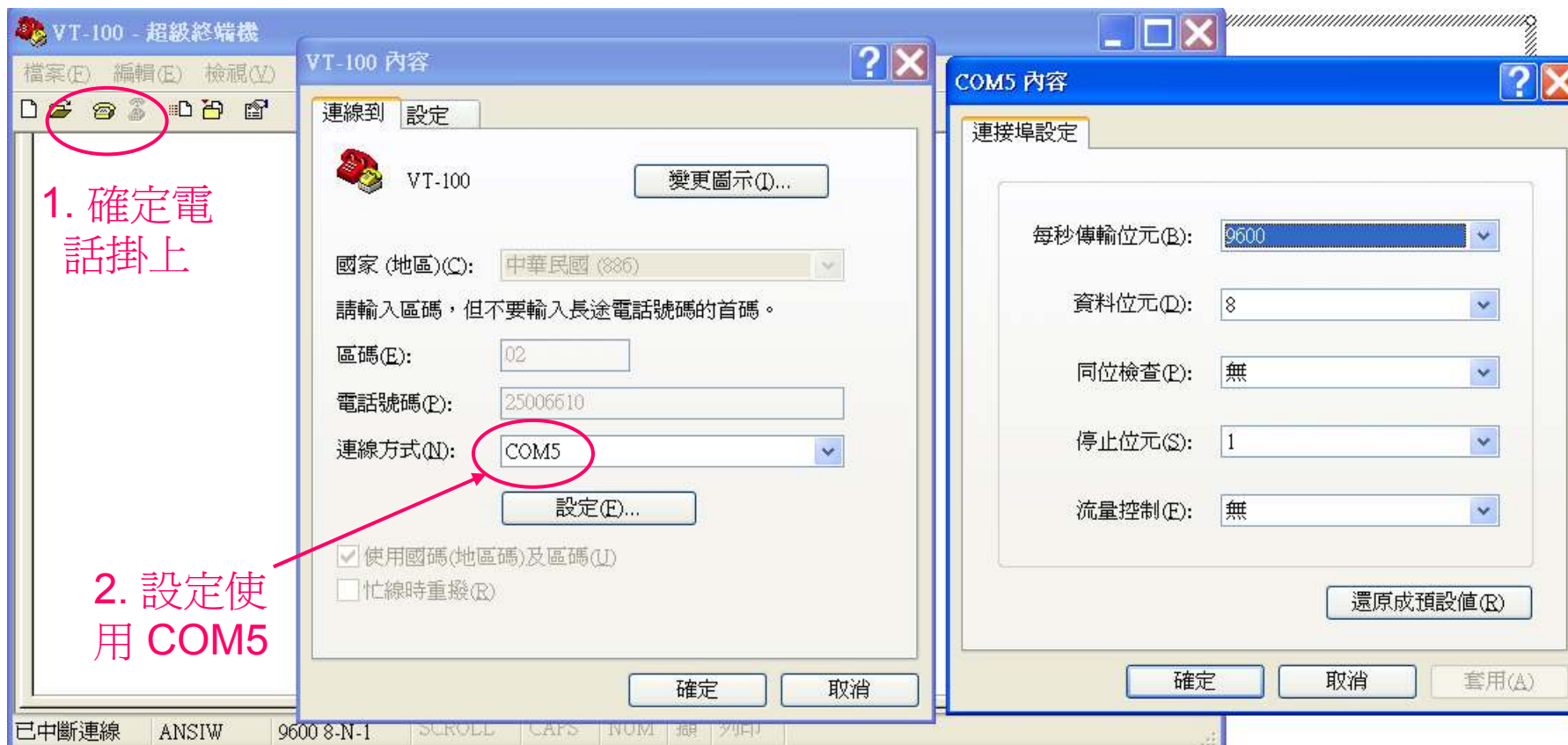
- 因現今筆記型電腦已經沒有 **RS-232** 介面
- **USB to RS-232** 轉接線/器 可以使用
- 在裝置管理員下，找出 **USB** 所模擬的 **COM Port**
 - 在下圖為模擬 COM5



設定超級終端機

- 開啓超級終端機

- 開始 → 所有程式 → 附屬應用程式 → 通訊
- 開啓“超級終端機”



練習 5-3

- 開啟” “\RTC\W401\Ans5-3\Ex5-3.mcp”
- 設定**USART**
 - 9600 , N , 8 , 1
 - 中斷設定: Disable TxD , Enable High Priority for RxD
- 送一字串“**MPLAB-C18 Workshop**”到終端機顯示
- **A/D** 轉換結果用 **PWM2** 控制 **LED** 亮度
- 終端機按“**p**” (小寫) **LED** 熄滅，並在終端機顯示字串：
“**Motor is Stopped !**”
- 終端機按“**s**” (小寫) **LED** 亮，並在終端機顯示字串：
“**Motor is Running !**”
- 終端機按“**r**” (小寫) 讀取馬達轉速控制值，並轉成十進制的**ASCII**在終端機顯示：
“**Current Motor Speed are : 1023**”



MICROCHIP

第六章

MPLAB-C18 函數庫

- 1. MPLAB-C18函數庫**
- 2. MSSP模組與I²C的函數庫**
- 3. 讀寫 I²C EEPROM**

了解 C18 函數庫

- 別忘了加入所使用函數庫的定義檔
(**#include <xxxx.h>**)
- 有了函數庫的支援，複雜的運算或資料處理較容易撰寫，可提高程式的功能及產品的附加價值
- 參考 **C:\MCC18\doc\hlpC18Lib.chm**
- **LIB** 檔案在 **C:\MCC18\LIB**
- **LIB** 原始檔案
 - Peripheral “C:\MCC18\src\pmc_common”
 - 其它 “C:\MCC18\src\traditional\xxxx”

C18 函數庫的分類

- 硬體週邊函數庫- **Hardware Peripheral Lib.**
- 軟體週邊函數庫- **Software Peripheral Lib.**
- 標準函數庫 - **General Software Library**
- 數學函數庫 - **Math Library**

- 自己的函數庫 - **Your Library**

硬體週邊函數庫

Hardware Peripheral Library

- A/D Converter Functions
- Input Capture Functions
- I²CTM Functions
- I/O Port Functions
- MicrowireTM Functions
- Pulse Wide Modulation (PWM) Functions
- Reset Functions
- SPITM Functions
- Timer Functions
- USART Functions
-

軟體週邊函數庫

Software Peripheral Library

- External LCD Functions
- Software I²C Functions (I/O Emulation)
- Software SPI Functions (I/O Emulation)
- Software UART Functions (I/O Emulation)
- External CAN2510 Functions

注意! 使用軟體模擬的函數庫時，需注意I/O腳位的設定，並在程式中加以修正

例: I²C腳位之定義是在"sw_i2c.h"中定義的

標準函數庫

General Software Library

- 字元分類函數
Character Classification Functions
- 數字與文字的轉換函數
Number and Text Conversion Functions
- 延遲函數
Delay Functions
- 記憶體和字串的函數
Memory and String Manipulation Functions
- 系統重置函數
Reset Functions
- 字元輸出函數
Character Output Functions

數學函數庫 Math Libraries

- **32-bit 浮點數數學函數庫**
32-bit Floating Point Math Libraries
- **ANSI C 標準數學函數庫**
C Standard Math Libraries & Functions

標準函數庫

字元分類功能(一)

- **isalnum()**: **unsigned char isalnum (unsigned char Data)**
 - 檢查輸入字元是否為字母或數字(A..Z , a..z , 0..9)
- **isalpha()**
 - 檢查輸入字元是否為字母(A..Z or a..z)
- **isctrl()**
 - 檢查輸入字元是否為控制字元(0x00..0x1F or 0x7F)
- **isdigit()**
 - 檢查輸入字元是否為數字(0..9)
- **isxdigit()**
 - 檢查輸入字元是否為十六進制格式(0..9 ,A..F ,a..f)

標準函數庫

字元分類功能(二)

- **isprint()**
 - 檢查輸入字元是否為可列印之字元(0x20..0x7E)
- **isspace()**
 - 檢查輸入字元是否為空白字元
 - “ ” 空格， “\t” TAB， “\r” CR， “\n” 換行， “\f” form feed， “\v” vertical tab ...
- **tolower()**
 - 將一大寫字母轉換成小寫字母
- **toupper()**
 - 將一小寫字母轉換成大寫字母
- **isgraph(), islower(), ispunct(), isupper()**

標準函數庫

數字與文字的轉換功能(一)

● **atob()** :

- 將指標指到的數字字串轉換成8位元的有號整數
- 例: “100”--> 0x64 , ”255”-->0xFF
- ”-128”-->0x80 , ”-2”-->0xFE

● **atoi()**

- 將指標指到的數字字串轉換成16位元的有號整數
- 例: “1000”--> 0x03E8=1000 , ”-1000”-->0xFC18=-1000

● **atol()**

- 將指標指到的數字字串轉換成32位元的有號整數
- 例: “1234567890”--> 0x0499602D2
- “-1234567890”--> 0xB669FD2E

標準函數庫

數字與文字的轉換功能(二)

● btoa()

- 將8位元有號整數轉換成數字字串後存到指標指到的位址
- 例: 0x80 --> "-128" , 100--> "100" , 199--> "-57"

● itoa()

- 將16位元有號整數轉換成數字字串後存到指標指到的位址
- 例: 0x1000--> "4096" , 1000--> "1000"

● ltoa()

- 將32位元有號整數轉換成數字字串後存到指標指到的位址

```
#include <p18c452.h>
#include <stdlib.h>

char string[12];
int code=-1000;

void main(void)
{
    itoa(code,string);
    while(1);
}
```


標準函數庫

數字與文字的轉換功能(三)

- **rand()** :產生 **0-32767** 的亂數
- **srand()** :產生虛擬的亂數
- **atof()** :指標指到的數字字串轉換成浮點值
- **tolower()** :將一字元轉換成大寫的字母
- **toupper()** :將一字元轉換成小寫的字母
- **ultoa()** :將32位元無號數轉換成數字字串後存到指標指到的位址

標準函數庫

延遲功能

- **Delay1TCY()** : 延遲一個 **Tcy**
- **Delay10TCY()** : 延遲一個 **10 Tcy**
- **Delay10TCYx()** : 延遲 **10 Tcy** 的倍數
- **Delay100TCYx()** : 延遲 **100 Tcy** 的倍數
- **Delay1KTCYx()** : 延遲 **1KTcy** 的倍數
- **Delay10KTCYx()** : 延遲 **10K Tcy** 的倍數

Tcy是指MCU指令執行的週期，以
4MHz的速度來說**Tcy=4MHz/4=1uS**

標準函數庫

記憶體和字串的操作(一)

- 比較兩個指標所指到的字串的內容
 - RAM 和 RAM 比較
 - memcmp (const void *buf1,const void buf2, mem_size)
 - ROM 和 ROM 比較
 - memcmppgm (const rom void *buf1,const rom void buf2, mem_size)
 - ROM 和 RAM 比較
 - memcmppgm2ram (const rom void *buf1,const void buf2, mem_size)
 - RAM 和 ROM 比較
 - memcmpram2pgm (const void *buf1,const rom void buf2, mem_size)

buf1: 要比較的陣列位址1 , buf2: 被比較的陣列位址2

mem_size: 陣列中有幾個byte要比較

回傳值: <0 (buf1<buf2) , ==0 (buf1=buf2) , >0 (buf1>buf2)

標準函數庫

記憶體和字串的操作(二)

- **strcpy()** : 複製字串(指標型態操作)
- **strcmp()** : 比較兩字串(指標型態操作)
- **strcat()** : 將一字串附加到另一字串
- **strchr()** : 尋找字串中的指定字元
- **strlen()** : 回傳字串長度
- **strlwr()** : 將字串中的大寫改爲小寫
- **strupr()** : 將字串中的小寫改爲大寫

更多的函數庫資料請參閱“C18 參考手冊”中
第九章“General Software Library”

可用的巨集指令

- 這些 **PIC** 的巨集指令可直接在**C18** 裡直接執行
- 底下的巨集指令定義在 **p18f4520.h** 檔的最後面

巨集指令	動作說明
Nop()	執行一個 NOP 指令
ClrWdt()	清除 Watch-Dog Timer
Sleep()	執行 SLEEP 指令，進入睡眠模式
Reset()	執行 RESET 指令，將 MCU 重置
Rlcf(Var)	將 Var 與進位旗號一起向左旋轉 ($C \leftarrow B7 \leftarrow B6 \dots B0 \leftarrow C$)
Rlncf(Var)	將 Var 向左旋轉(不含進位旗號) ($B7 \leftarrow B6 \dots B0 \leftarrow 0$)
Rrcf(Var)	將 Var 與進位旗號一起向右旋轉 ($C \rightarrow B7 \rightarrow B6 \dots B1 \rightarrow B0 \rightarrow C$)
Rrncf(Var)	將 Var 向右旋轉(不含進位旗號) ($0 \rightarrow B7 \rightarrow B6 \dots B1 \rightarrow B0$)
Swapf(Var)	將 Var 高、低4位元互換

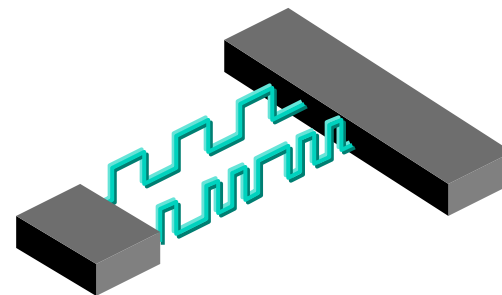
同步串列通訊(一)

Master Synchronous Serial Port (MSSP)

可工作於SPI或 I²C 兩種模式之一

●SPI (Serial Peripheral Interface) 模式

- 可程式化速率設定
- 最高速率 (@ 40 MHz) :
 - Master 10.0 Mbps
 - Slave 4.29 Mbps
- 可程式化設定接收、發送的時脈(clock)動作極性
- 支援兩種傳輸模式 Microwire™ 和 Motorola SPI



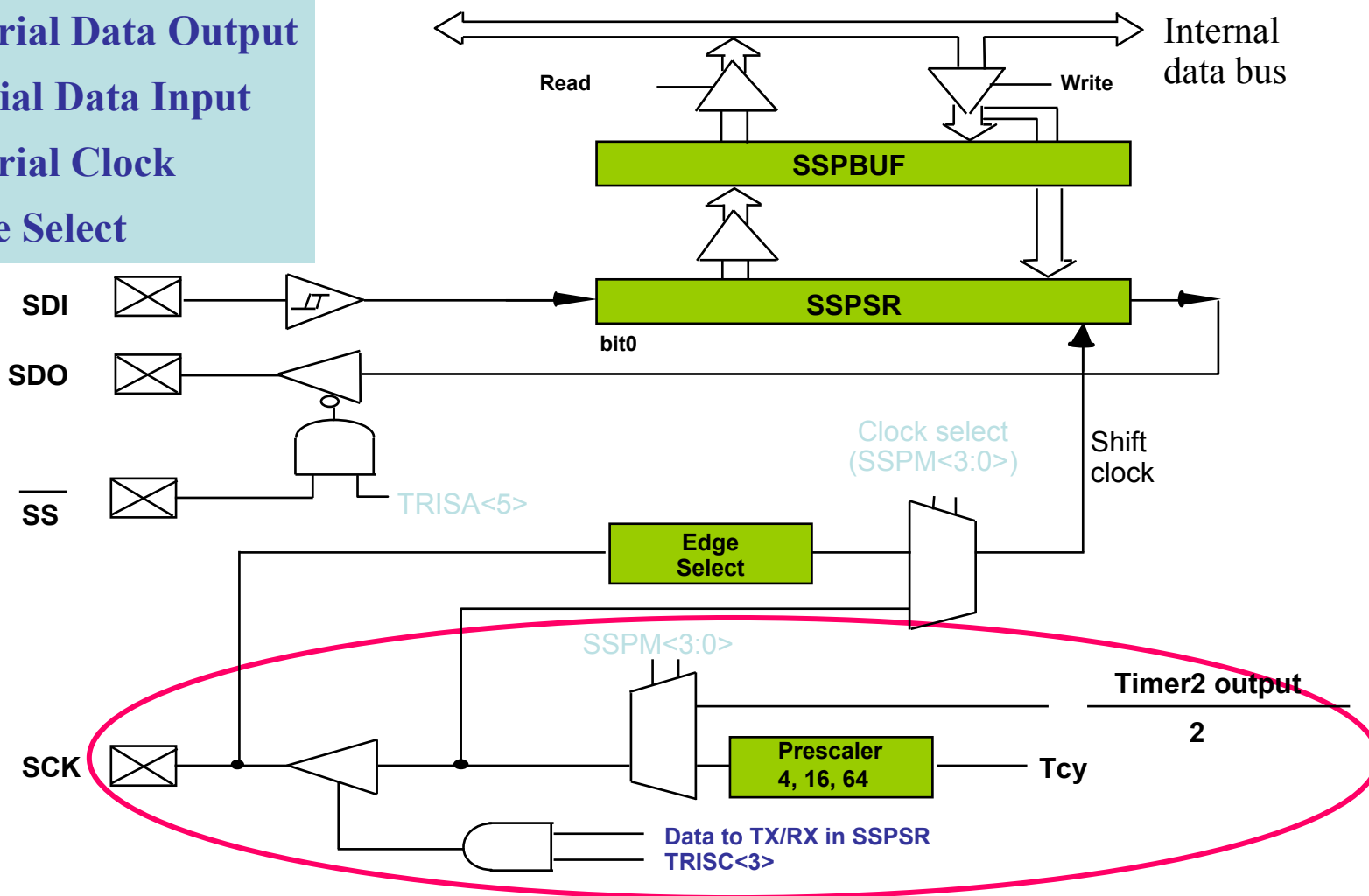
SPI 方塊圖

SDO: Serial Data Output

SDI: Serial Data Input

SCK: Serial Clock

\overline{SS} : Slave Select



Master Mode: SCK is the clock output pin

SPI 函數庫

- `OpenSPI()` 設定 SPI 工作模式
- `DataRdySPI()` 是否有接收資料在SSPBUF暫存器?
- `getsSPI()` 從SPI讀取一字串
- `putsSPI()` 寫一字串到SPI
- `ReadSPI()` 從SPI讀取一字元
- `WriteSPI()` 寫一字元到SPI
- `CloseSPI()` 關閉 SPI 介面

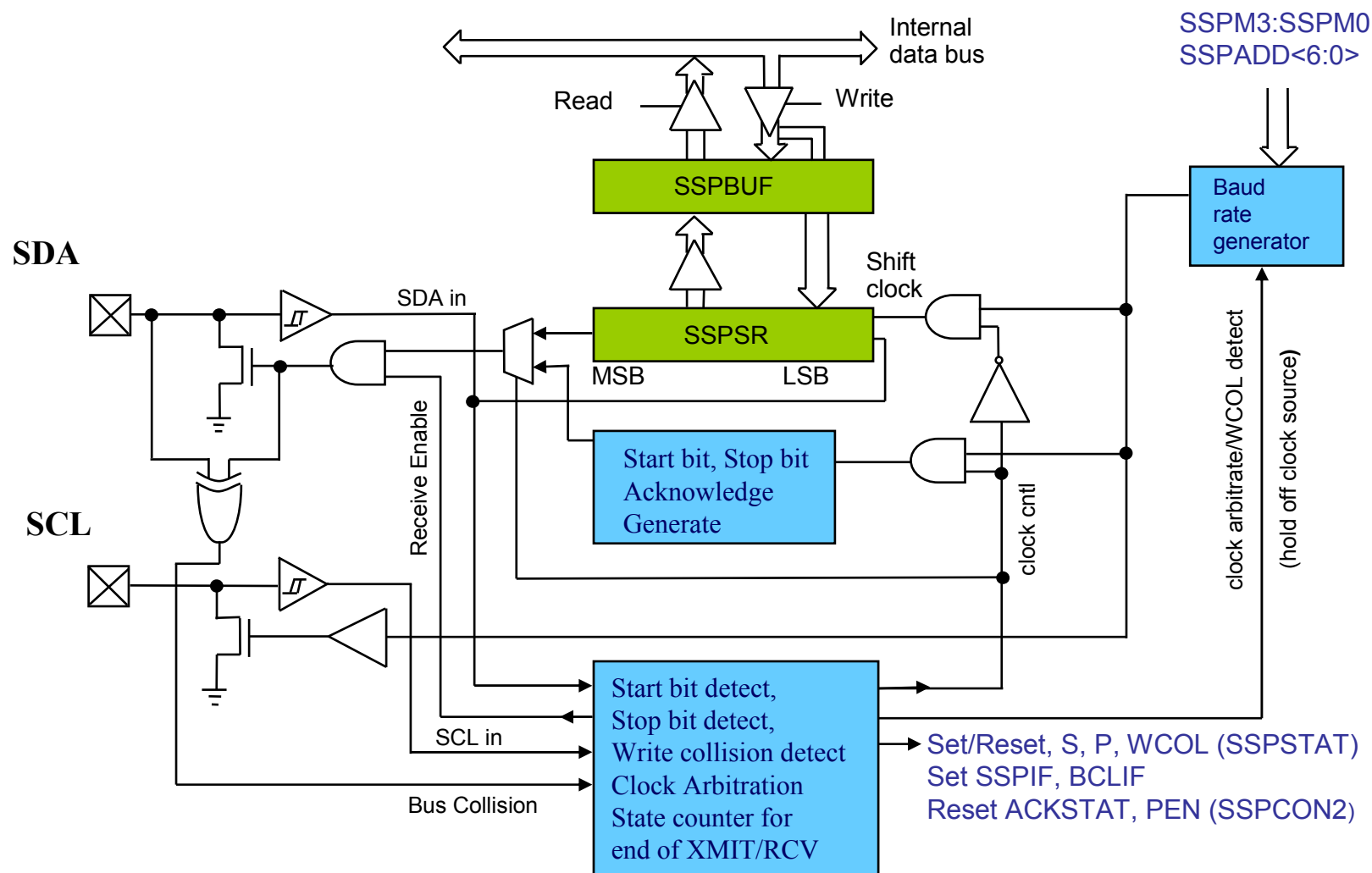
詳細 SPI 的函數操作功能及使用方式，請參閱
“<C:\MCC18\doc\periph-lib\SPI.htm>”

同步串列通訊(二)

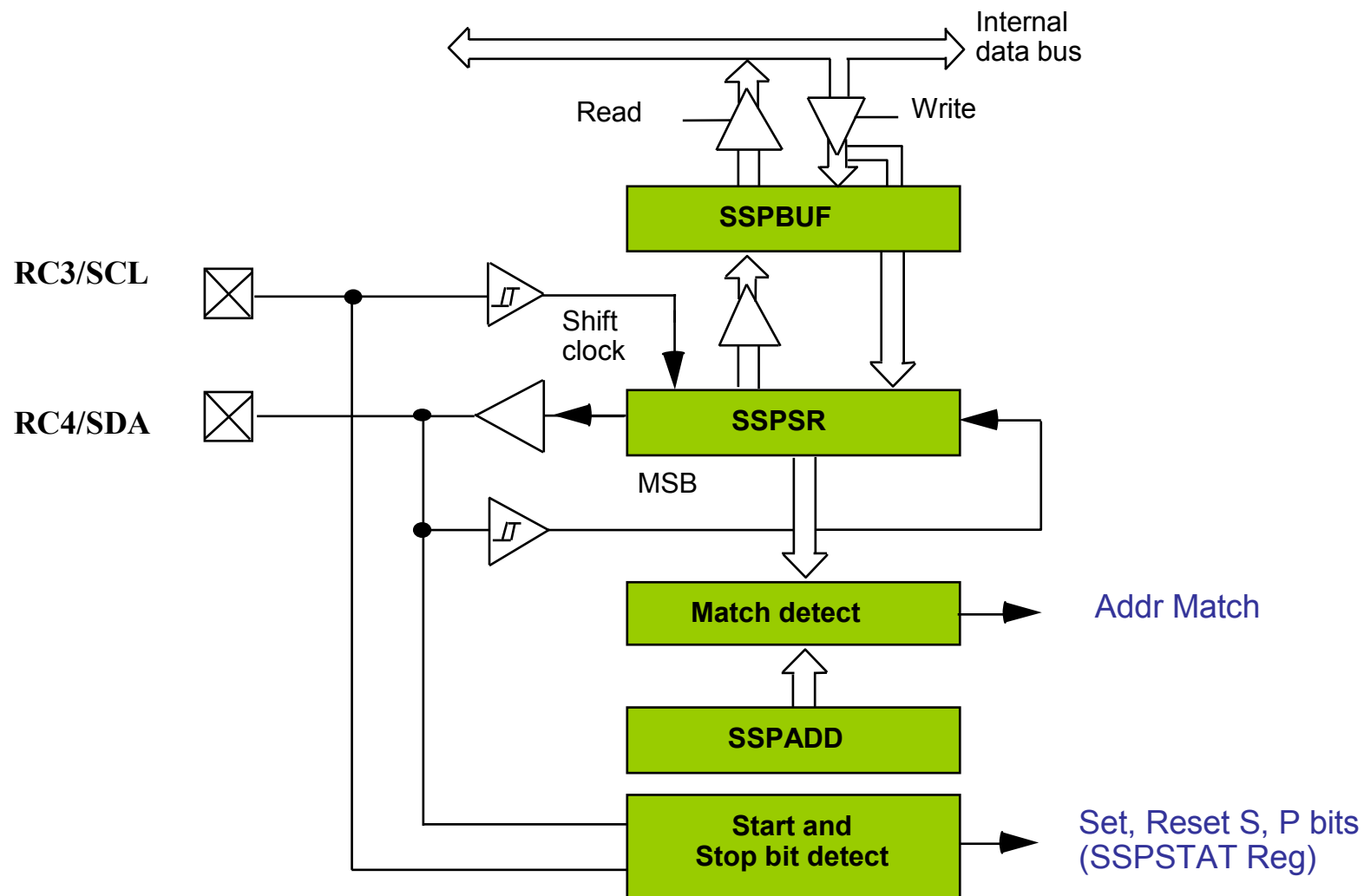
Master Synchronous Serial Port (MSSP)

- **I²C (Inter-Integrated Circuit) 模式**
 - 全硬體化設計，支援下列三種I²C的操作模式
 - Master Mode
 - Multi-master Mode
 - Slave Mode
 - 支援 7-bit 或 10-bit 位址模式
 - 傳送及接收速度:100kHz， 400kHz， 1 MHz
 - 支援通用位址 (General call) 模式“address=0x00”

I²C Master 模式方塊圖



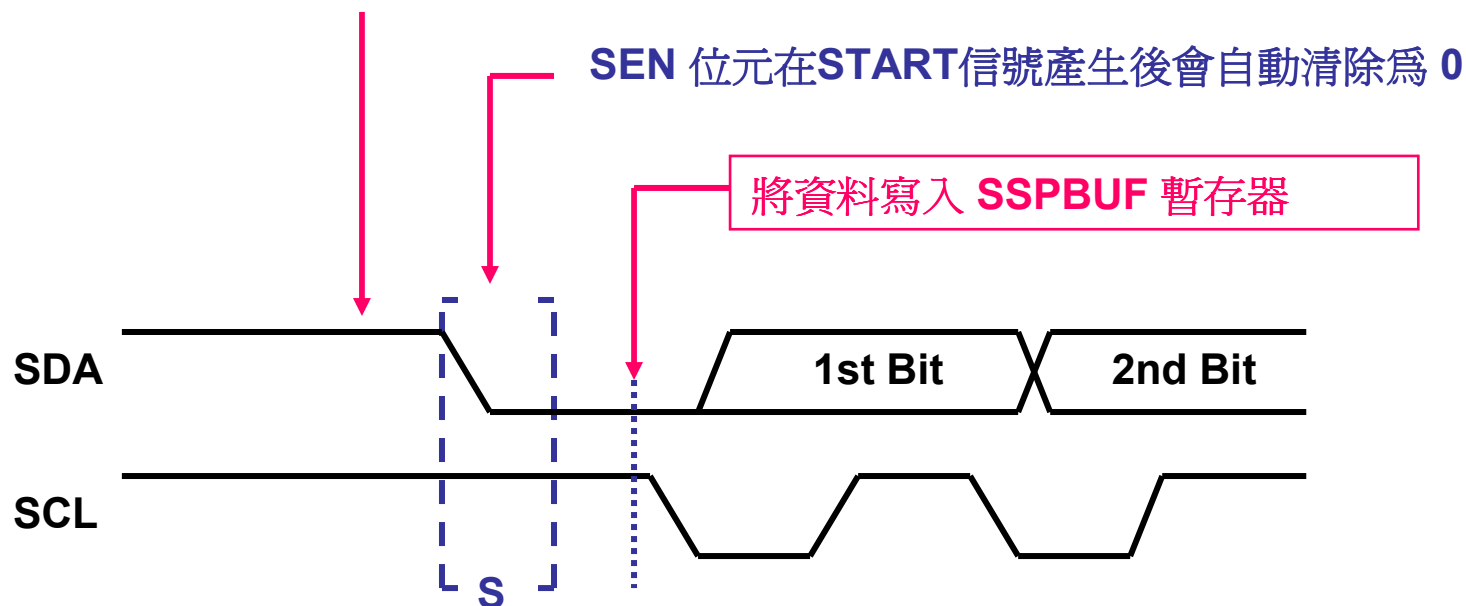
I²C Slave 模式方塊圖



I²C Start Condition 時序

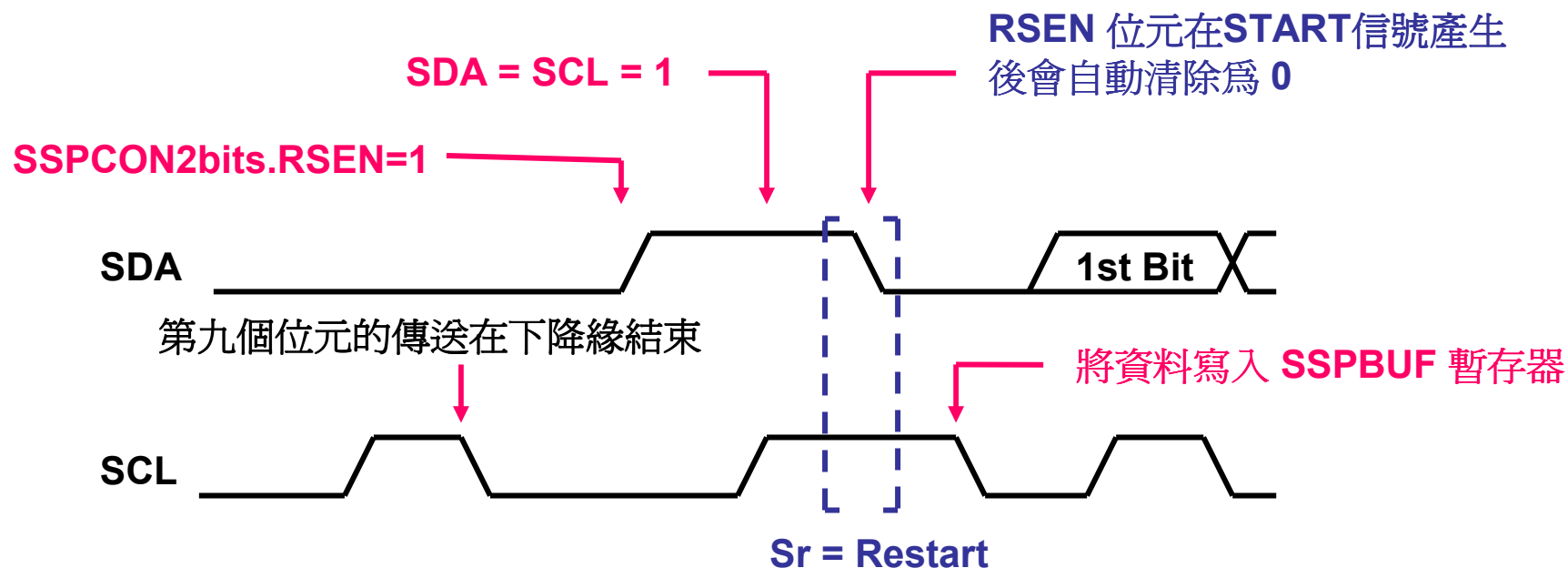
- **Bus Ready** : SDA和SCL長時間都在Hi時
- 一般而言，在SCL為Hi時SDA是不能有位準上的改變
變
 - (除了Start & Stop 條件外)
- **Start** : 在SCL為Hi，SDA由Hi變Low時

SSPCON2bits.SEN=1 (要求產生START信號)



I²C Restart Condition 時序

- **Restart** : 在第九個位元(**ACK**)傳送結束後，無須送出**STOP**訊號直接將**SCL**和**SDA**拉成**Hi**電位後再產生**START**訊號



I²C ACK 回應訊號說明

■ 每一位元的傳送都在SCL下緣時結束

■ **Slave 回應 Master** 以確定已接收到位址或資料

- 位址比對正確時
- 收到正確資料
- Slave回應ACK位元，0表示接收正確，1為錯誤

■ **Master**可以檢查 SSPCON2bits.ACKSTAT以了解
Slave回覆的ACK狀態

■ **Master**回應Slave以要求Slave繼續傳送資料

- Master回應ACK位元，0表示繼續傳送，1為結束傳送

■ **Master**可以設定SSPCON2bits.ACKDT再啟動
SSPCON2bits.ACKEN 位元來知會Slave

I²C ACK回應訊號時序

以Master回應Slave為例說明

① 設定 SSPCON2bits.ACKDT

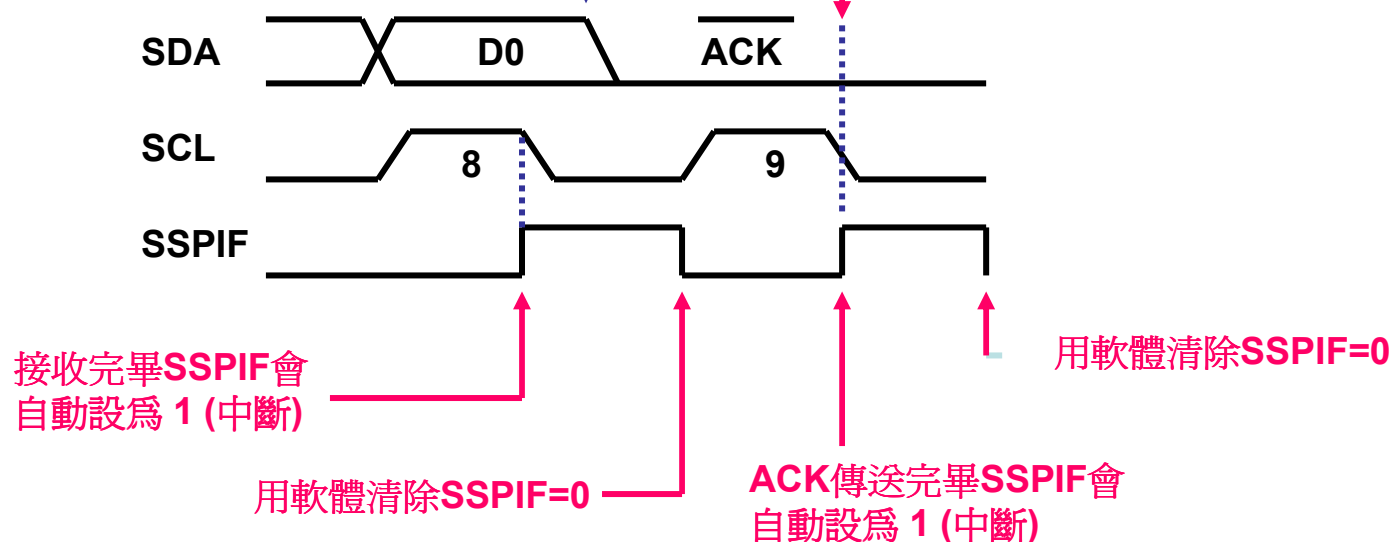
➤ = 0 Acknowledge , =1 Not Acknowledge

② 在設定 SSPCON2bits. ACKEN =1 以啟動ACK的傳送

Acknowledge sequence starts here

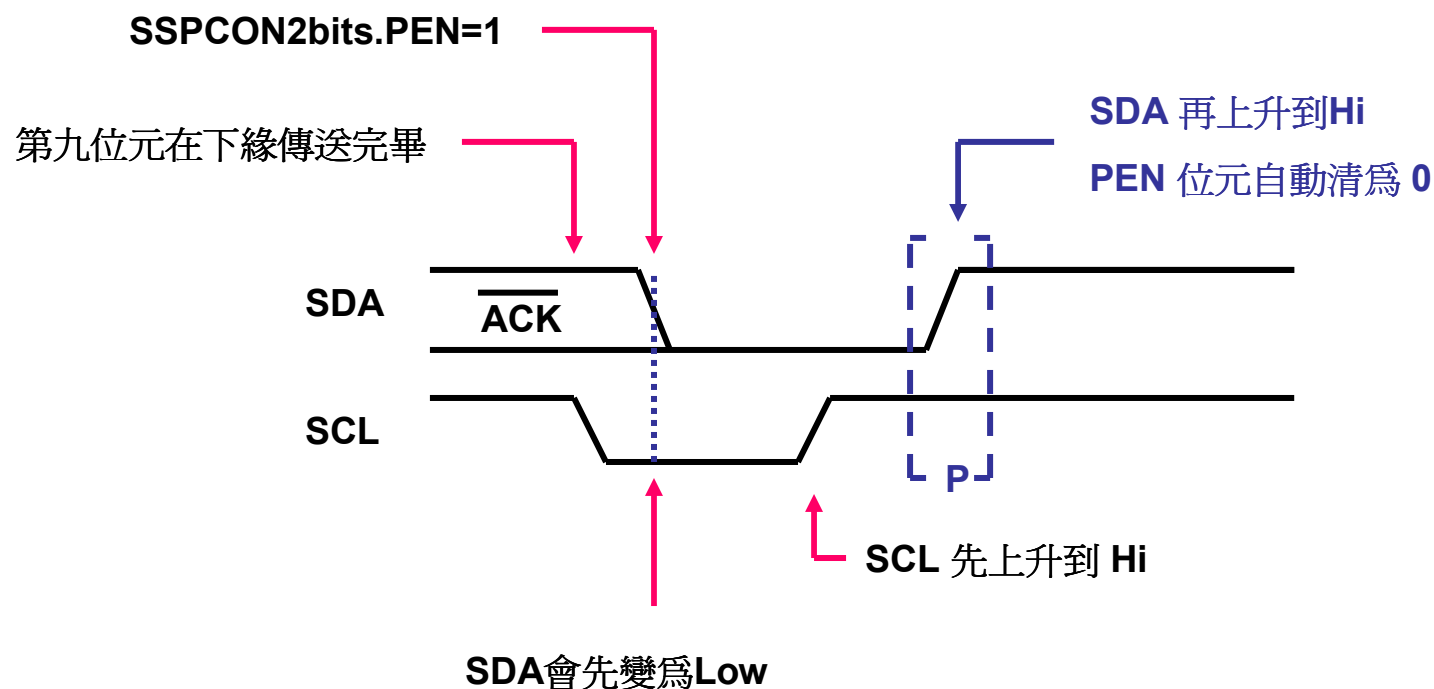
Write to SSPCON2
ACKDT = 0 , ACKEN = 1

ACKEN 自動清除為 0

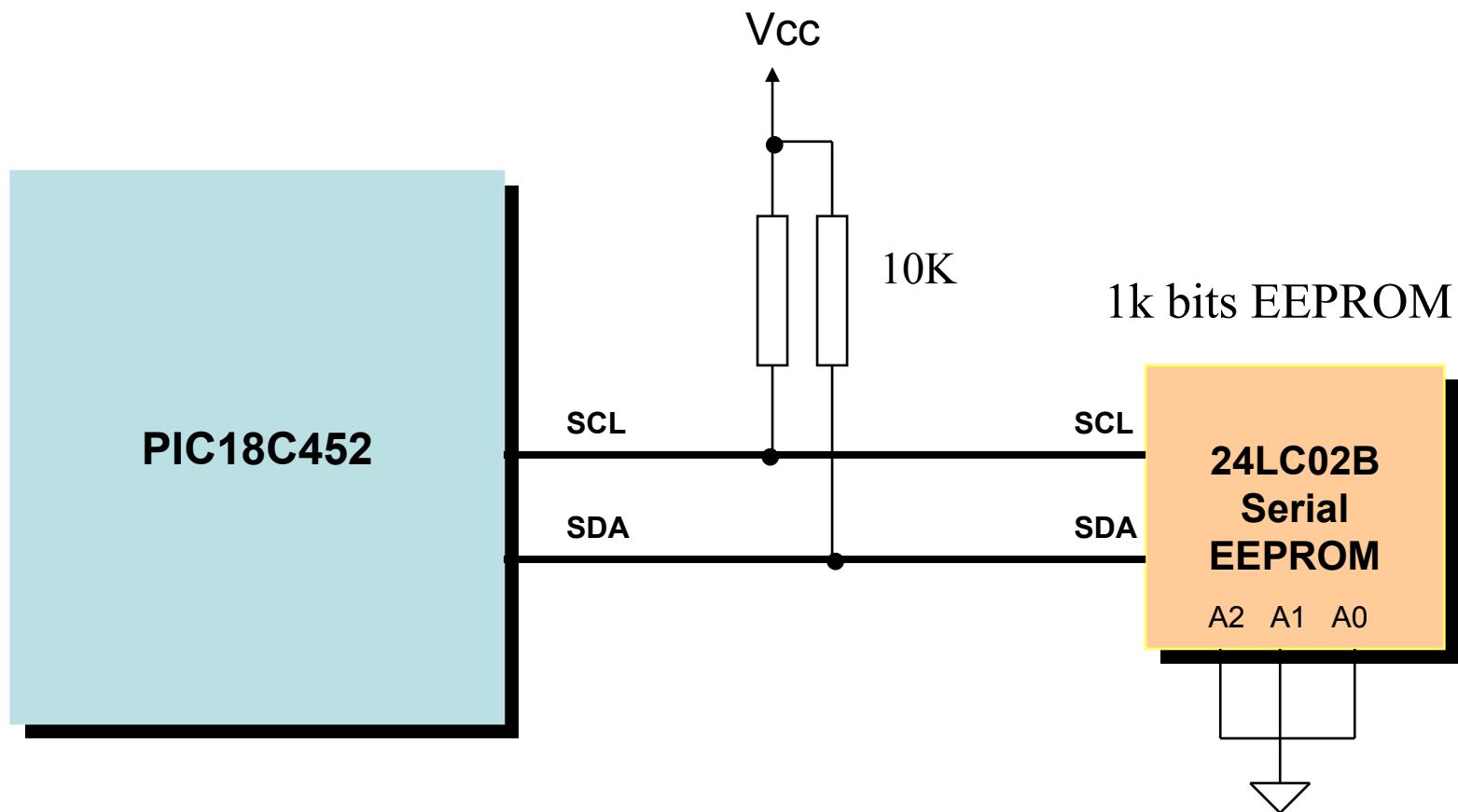


I²C Stop Condition 時序

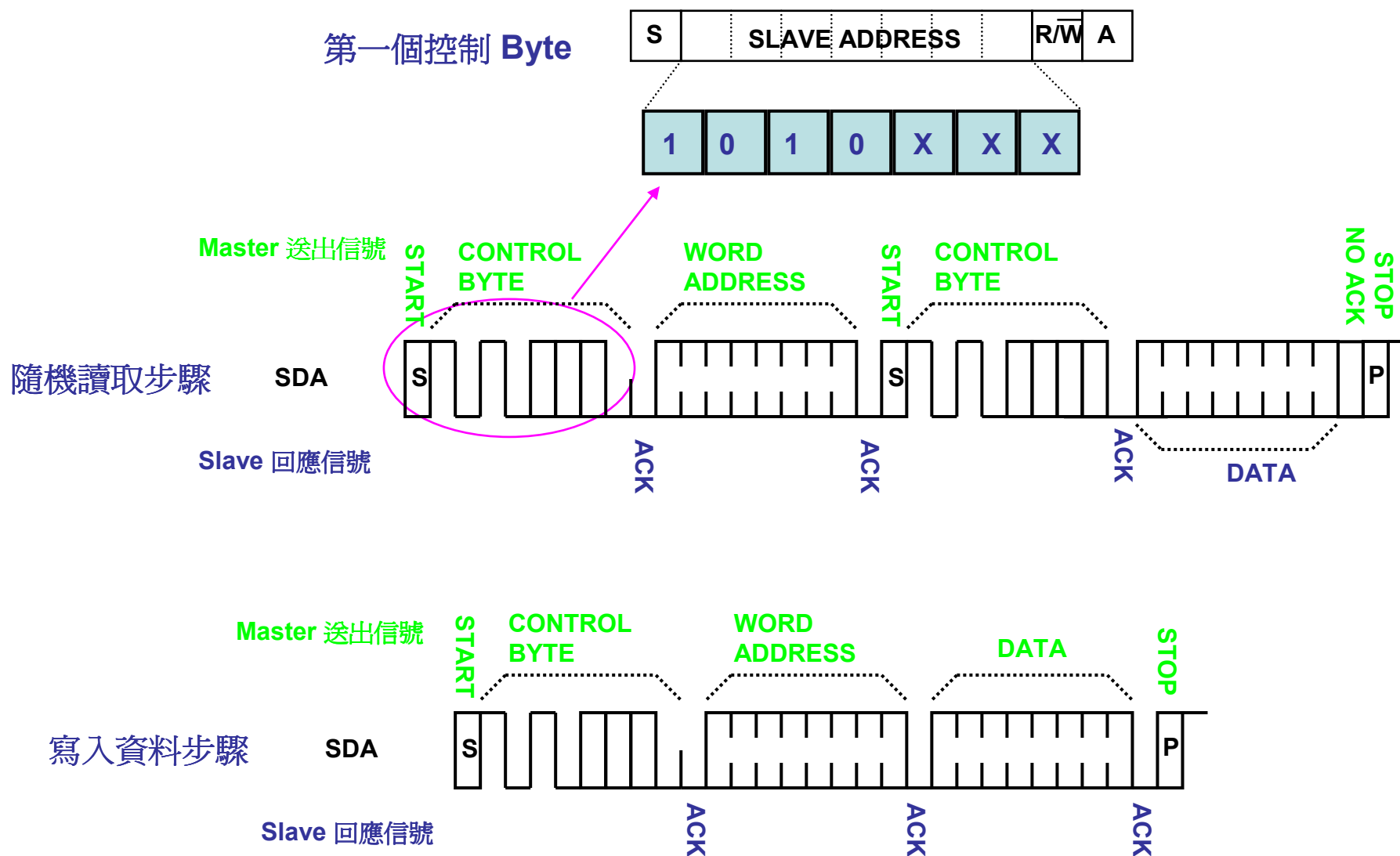
- 產生方式 : SSPCON2bits.PEN=1
- **Master**用來宣告本次的I²C的傳送結束



I²C EEPROM 基本接線圖



24LC02B 基本時序圖



I²C 函數庫的版本問題

- **C18** 在支援 **I²C** 的函數庫有分成數個版本，使用之前須知道該 **I²C** 被歸類在那個版本裡。
- 有關版本的歸類要參考：
 - C:\MCC18\doc\periph-lib\I2C.htm 的檔案說明
 - PIC18F4520 是屬於 I2C_v1 的設定
- 例如：單是 **OpenI2C()** 就有三種，用那一個才是正確的？
 - OpenI2C() , OpenI2C1() , OpenI2C2()

I²C Serial EEPROM 函數庫

底下 I2C EEPROM 函數適用於 24LC00 ~ 24LC16 的元件使用
24LC32 (含) 以上的元件需自行改寫成 16-bit Address 的函數

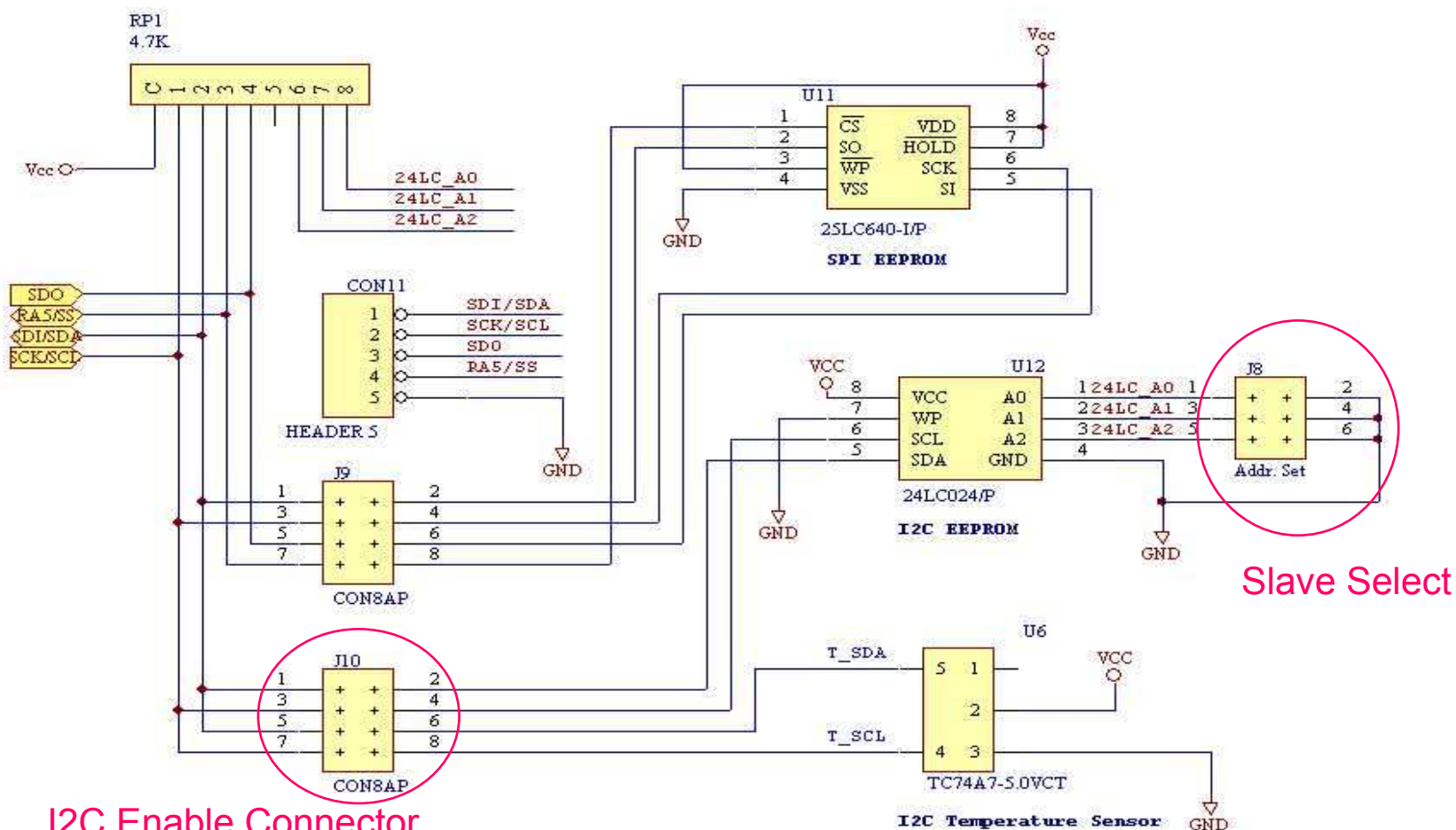
- **EEByteWrite** (控制碼，位址，寫入的資料)
 - 將一Byte的資料寫到指定的位址上
- **EECurrentAddrRead** (控制碼)
 - 讀取目前位址上的資料
- **EEPageWrite** (控制碼，位址，寫入資料的指標)
 - 將一字串寫到指定的位址上，直到字串結束(null)
- **EEAckPolling** (控制碼)
 - 檢查 EEPROM 工作狀態
- **EERandomRead** (控制碼，位址)
 - 讀取指定的位址上的資料
- **EESequentialRead** (控制碼，位址，資料存放的指標，長度)
 - 將讀取一字串並存入到指定的RAM位址

24LCxx 晶片位址選擇

- **APP001 板子上 J8 作為 I2C EEPROM Device Address 的設定**
- 使用時請將 **J8** 三個位址設定都短路
 - 設定 A0, A1 及 A2 接腳為 Low
 - 24LC02B 這三隻腳是沒有作用的
 - 24LC32A 這三隻腳是有作用的
- 這三支接腳會因元件的不同而易，使用其他編號的元件時須查明其 **Data Sheet**
- **J10 短路 (1&2, 3&4)，接通 I2C bus 到 24LC02B 的 SCL & SDA 腳**

APP001 I²C 線路

- APP001 板子上有兩個並聯的 I²C 元件
 - 24LC02B 及 TC74-A7 溫度感測器
 - 因為 I2C Slave Address 不一樣，不會打架



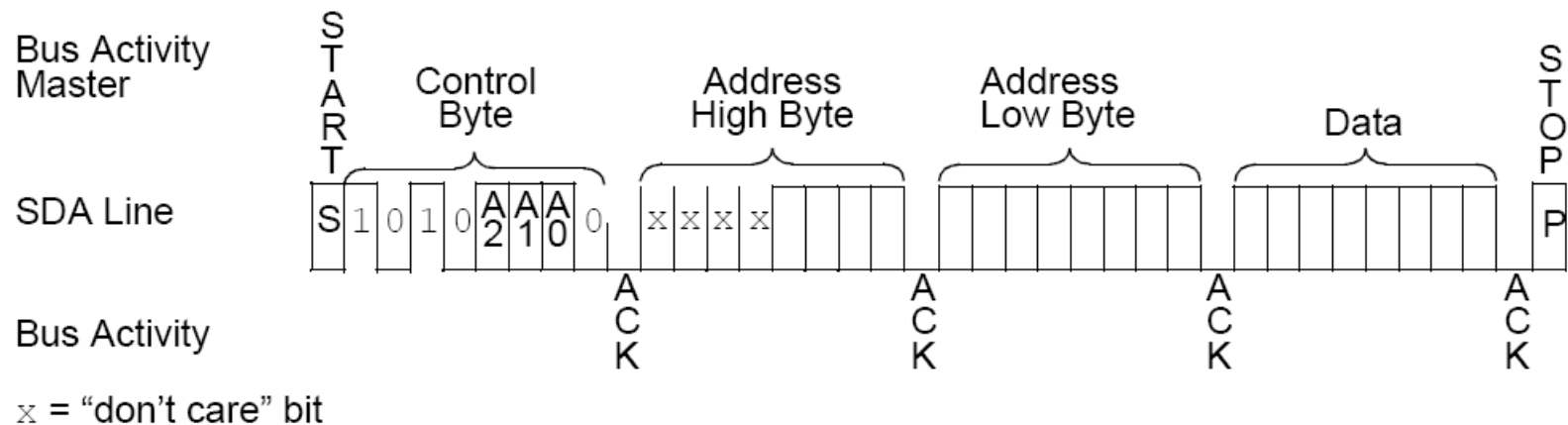
I2C Enable Connector

練習 6-1 (存取 24LC02B)

- 開啓“\RTC\W401\Ans6-1 \Ex6-1 .mcp”
- 利用 **EEPROM** 的函數庫來操作 **24LC02B** 的讀寫
 - PIC18F4520工作在 Master模式
 - 速率= $F_{osc}/4/(SSPADD+1)$; $16\text{MHz} / 4 / (39 + 1) = 100\text{KHz}$
- 利用終端機輸入至少**4**個數字並用“**Enter**”鍵結束
 - 最多可輸入**16** 個數字，超過時自動跳出輸入程序
 - 程式會檢查輸入的是否為數目字
 - 若輸入數字少於 **4** 個數字，按“**Enter**”無效
 - 輸入的數字存入**EEPROM**中，並於下一個位址補存一個“null”以符合字串的格式，起始位址為 **0x0000**
- 數字輸入完成後，自 **EEPROM** 讀取剛存入的字串並顯示在 **LCD** 的螢幕上

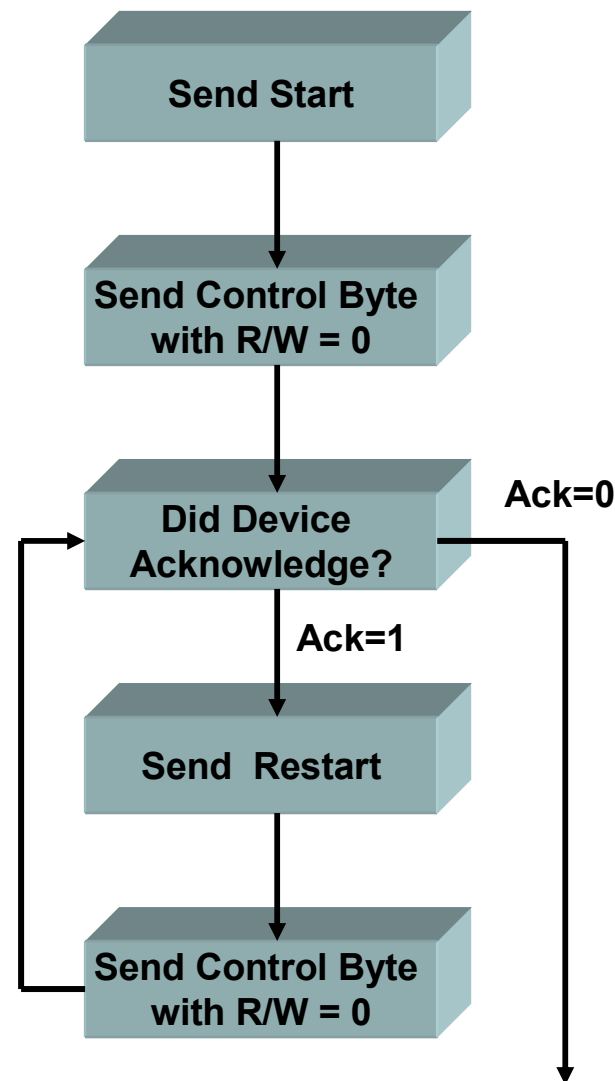
練習 6-1A (存取 24LC32A)

- 開啓”\RTC\W401\Ans6-1a\Ex6-1a.mcp”
- **24LC32A** 爲 **32K bits** 的容量，需 **4K Bytes** 的位址線來定址
 - 0x0000 ~ 0x0FFF 需 A0 ~ A11
- 所以 **Address** 需要 **Hi-Byte + Low-Byte**

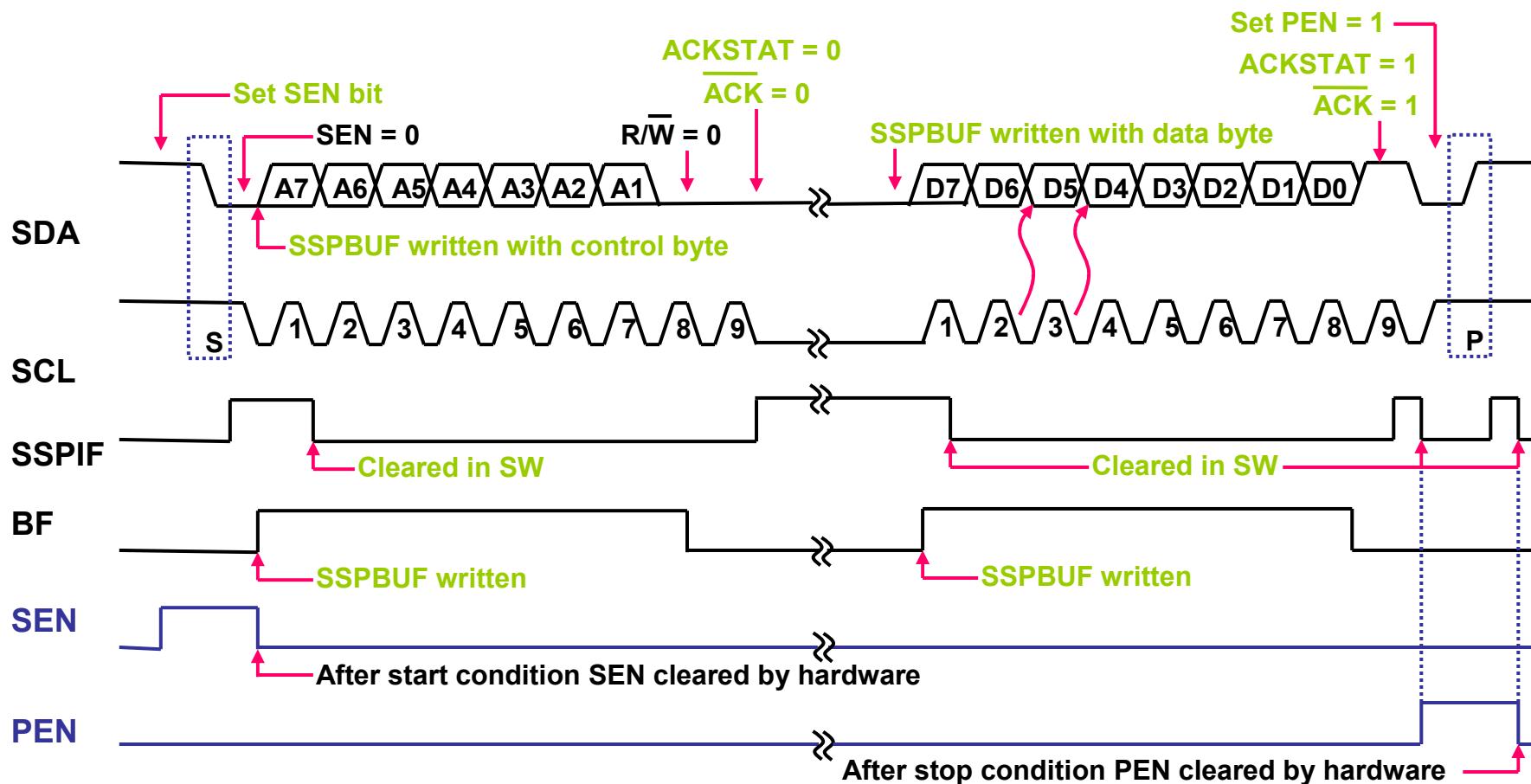


24LCxx Ack 信號的偵測

- 24LCxx 在寫入時間不會回應ACK信號 (ACK=1)
- 偵測ACK信號的回應以得知是否進行下一次的讀、寫
- 如果你想要正確的偵測Ack以決定是否可對EEPROM做下一個動作時，右邊流程圖是必須的



I²C Master 模式下 發送資料的時序



注意是不是每個動作完成後，SSPIF 都會被設為1？

I²C 函數庫

- `OpenI2C()` 設定 I²C 的工作模式
- `DataRdyI2C()` 是否有接收資料在 SSPBUF ?
- `StartI2C()` 產生 Start 信號
- `StopI2C()` 產生 Stop 信號
- `IdleI2C()` 產生等待信號直到 Bus 閒置
- `RestartI2C()` 產生 Restart 信號
- `WriteI2C()` 寫入一Byte到 I²C Bus
- `ReadI2C()` 從 I²C Bus 讀取一Byte
- `getI2C()` 從 I²C Bus 讀取一字串
- `putI2C()` 寫入一字串到 I²C Bus
- `AckI2C()` 送出一個 ACK
- `NotAckI2C()` 送出一個 NACK
- `CloseI2C()` 關閉 I²C

Ex6-2 程式說明

共用 24LC02B & 24LC32A 同一程式

```
//#define EEPROM_24LC32A ( 只要刪掉註解後就成 24LC32A 的程式 )
:
void EEPROM_Write(unsigned char ctrl,unsigned char addr,unsigned char data)
{
    IdleI2C();                // ensure module is idle
    StartI2C();               // Start condition
    I2C_Done();               // Wait Start condition completed and clear SSPIF flag

    WriteI2C(ctrl);           // Write Control+Write to EEPROM & Check BF flag
    while(SSPCON2bits.ACKSTAT); // wait until received the Acknowledge from EEPROM
    I2C_Done();               // Clear SSPIF flag

    #ifdef EEPROM_24LC32A      //如果有定義使用 24LC32A 就多送一個 address byte
    WriteI2C(0x00) ;
    while(SSPCON2bits.ACKSTAT) ;
    I2C_Done() ;
    #endif

    WriteI2C(addr);           // Write Address to EEPROM
    while(SSPCON2bits.ACKSTAT); // wait until received the Acknowledge from EEPROM
    I2C_Done();

    WriteI2C(data);           // Write Data to EEPROM
    while(SSPCON2bits.ACKSTAT); // wait until received the Acknowledge from EEPROM
    I2C_Done();

    StopI2C();                // Stop condition
    I2C_Done();               // Wait the Stop condition completed
}
```

Ex6-2

- 開啓“\RTC\W401\Ans6-2\Ex6-2.mcp”
- 注意一下 **APP001** 是用那一顆 **I²C EEPROM** ？
- 改寫 **Ex6-1** 的程式，將程式中 **EEByteWrite ()** 及 **EEAckPolling ()** 函數庫改爲 **I²C** 的基本函數庫。
- **WriteI2C ()** 函數在送出一筆資料後會檢查 **PIR1bits.SSPIF** 旗幟，以判斷發送是否完成
- **ReadI2C ()** 函數在啓動接收暫存器後會檢查 **PIR1bits.SSPIF** 旗幟，以判斷接收是否完成



第七章

合併使用 **C18** 與 **Assembly**

1. 使用嵌入式組合語言
2. **C18** 與 **MPASM** 合用
3. 用組合語言完成標準**C**函數

使用嵌入式組合語言

In-Line Assembly

用C寫為何還需組合語言

- 最主要目的 - 提高執行速度，增加程式對事件處理的效率
- C 語言對位元操作不及組合語言方便
 - 其實使用 C18 兩者是差不多
- 精確時序控制
 - IR Transceiver, Remote Timing...
- 程式塞不下MCU時
- 時間很多 (用 C 完成的關係)
- 因為不會用 C 寫

C18 的嵌入式組合語言組譯器

(一)

- 嵌入式組合語言稱為
 - In-Line Assembly
- 語法像簡易的 **MPASM** (限制)
 - 不支援虛指令(directive)
 - **ORG, EQU, RES, BANKSEL, SET, #DEFINE**
 - 簡單的語法，不支援標準定義
 - 無 xxx.inc 檔案中的定義：**RCIF,ADIF,W,F,FAST**
- **_asm** 來宣告使用嵌入式組合語言
- **_endasm** 作為嵌入式組合語言的結束
- 基本常數定義是採十進制
 - **movlw 10 ==> movlw 0x0a**

C18的嵌入式組合語言組譯器 (二)

- 還好它可支援
 - 它可以看到函數名稱
 - 它可以使用C的變數
 - 標記(Label)
 - 標記必需用冒號分辨 (Tx_Loop:)
 - 暫存器定義的名稱
 - 與MPASM相同 (RCREG, SSPBUF ..)
- 一些有關指標的暫存器最好不要動它
 - FSR0 , FSR1 , FSR2
 - PCLATU , PCLATH , TBLPTRx

PIC18Fxxxx 嵌入式指令寫法

MOVF f,d,a (d 和 a 只能用 1 或 0 來表示)



d = Destination Bit
d = 0 for destination W
d = 1 for destination F

a = Access Bit
a = 0 for Access Bank
a = 1 for bank per BSR

f = 8-bit Register Address

一般在MPASM寫法：

```
movf    PORTD,W
andlw   b'11110000'
xorwf   Temp_Var1,W
btfss   STATUS,Z
goto    DiscFail
bsf     STATUS,C
return
```



In-Line Assembly寫法：

```
movf    PORTD,0,0
andlw   0xf0
xorwf   Temp_Var1,0,0
btfss   STATUS,2,0
goto    DiscFail
bsf     STATUS,0,0
return  0
```

Ex7-1 嵌入式組合語言範例

```
#pragma interrupt isr_high
```

```
void isr_high(void)
```

```
{
```

```
    _asm
```

```
// 使用嵌入式組合語言的宣告
```

```
    movff RCREG,Rec_Data    // move USART data to
```

```
    // Rec_Data & Clear RCIF
```

```
    movlw '='               // Load "=" to Wreg.
```

```
    movwf TXREG,0          // Send to Terminal
```

```
    nop
```

```
Tx_Loop:    btfss PIR1,4,0    // Check TXIF rising to Hi
```

```
            goto Tx_Loop    // Is Low, loop test
```

```
            movff RCREG,TXREG // Send back to Terminal
```

```
    _endasm
```

```
// 結束使用嵌入式組合語言
```

```
}
```

```
#pragma code
```

Ex7-1

嵌入式組合語言寫法

- 開啓“\RTC\W401\Ans7-1 \Ex7-1 .mcp”
- 利用 **Ex6-1a** 程式中的中斷服務程式改用 **C18** 的內建組合語言方式撰寫
 - 程式可參考前頁的範例
 - 接收到UART資料後，並傳回“=”及接收資料給終端機
- 這種嵌入式組合語言的寫法在功能上是否有差異

C18 與 MPASM 合用

Mixing C and Assembly

C18 呼叫組合語言 (變數的傳遞)

- **區域變數(Local)**：是在函數內部所宣告的變數，其視野僅在本函數內，其它的函數(包括組合語言)是無法使用的
- **公用變數(Global)**：是在函數以外的地方宣告，實際佔有記憶體體的變數，每個函數(包括組合語言)都可以使用它
- 當組合語言程式被C語言呼叫時，當然是用公用變數做資料的傳遞
- 組合語言的寫法是可重新定址(**Re- Locatable**)的語法，由**MPLINK**安排變數與程式實際位址
- 正確的對變數宣告為 **EXTERN** 或 **GLOBAL**

C18 呼叫組合語言

(Global - Assembly Only)

● GLOBAL的變數

- 語法: GLOBAL <變數名稱1>,<變數名稱2>,<...>
- Global 是用來宣告此變數是公用型態，可被其它的程式使用
- 適用範圍: 組合語言內宣告的變數給C18及其它的組合語言來使用

● GLOBAL的程式

- 語法: GLOBAL <副程式名稱1>,<副程式名稱2>,<...>

● Global 是用來宣告此副程式是公用程式，可被其它的程式呼叫使用

- 適用範圍: 宣告組合語言副程式給C18及其它的組合語言來呼叫使用

C18 呼叫組合語言 (Extern in Assembly)

● EXTERN的變數

- 語法: EXTERN <變數名稱1>,<變數名稱2>,<...>
- Extern 是用來告訴組譯器(MPASM)此變數是由別的程式所宣告的，連結器(MPLINK)會決定其位址

● EXTERN的程式

- 語法: EXTERN <副程式名稱1>,<副程式名稱2>,<...>
- Extern 是用來告訴組譯器(MPASM)此副程式名是在別的程式裡，由連結器(MPLINK)來安排其呼叫位址

C18 呼叫組合語言 (Extern in C18)

● EXTERN的變數

- 語法: EXTERN <變數型別> 變數名稱;
 - extern unsigned char Var1;
 - extern near unsigned char Var1;
- Extern 是用來告訴編譯器(Compiler)此變數是由別的程式所宣告的，連結器(MPLINK)會決定其位址

● EXTERN的函數(在C程式裡，僅需提供被呼叫函數的原型(**prototype**)宣告即可)

C18 呼叫組合語言 (可重新定址的語法)

● 程式

- 不使用“ORG”直接定位址，改由“CODE”宣告為程式節區

● 變數

- 用“UDATA”或“UDATA_ACS”宣告為變數節區
- 不使用“CBLOCK”定變數區,改由“RES”
- 變數最好安排在ACCESS BANK裡
- 對不是ACCESS BANK內的變數，善用“BANKSEL”來設定適當的 BSR 暫存器(誰曉得C18將變數擺在哪裡)
- 對公用變數宣告，C與組合語言之中只能有一個宣告
 - 重複宣告將被視為兩個不同的變數，造成變數無法傳遞

範例 7-2 (Ex7-2.c)

- **Rec_Data** 為 **C** 底下所宣告的公用變數，用來傳回中斷服務程式所接收的 **RS-232** 資料
 - unsigned char Rec_Data
- **Var1** 為 **ISR.asm** 下所宣告的公用變數在 **Access Bank** 內，所以在 **Ex7-2.c** 中宣告成 **extern**，也可以用來傳遞資料
 - extern near unsigned char Var1
- **ISR_Routine** 外部函數的原型宣告
 - void ISR_Routine(void);

範例 7-2 (ISR.asm)

- **Rec_Data** 為 **extern** 公用變數，用來傳遞中斷服務程式所接收的 **RS-232** 資料
 - `extern Rec_Data`
- **Var1,Var2** 為 **ISR.asm** 下所宣告的公用變數在 **Access Bank** 內，也可以用來傳遞資料 (**C** 看的到)

```
global      Var1,Var2
            udata_acs
Var1        res      1
Var2        res      1
```

- **ISR_Routine** 被宣告成公用程式 (給 **Ex7-2.c** 呼叫)
 - `global ISR_Routine`

Ex7-2

使用 MPASM 的組合語言

- 開啓“\RTC\W401\Ans7-2 \Ex7-2 .mcp”
- 記得用 “**Edit Project**” 加入組合語言檔 **ISR.asm**
 - 此時有 Ex7-2.c、I2C_EEP_SUBS_24LC32A.C、ISR.asm 及 WAP_LCD.c 等程式檔在 project 中
- 利用“**Watch Window**”觀察變數的位址及執行後的內容
 - Var1 , Var2 是否在 Access Bank ?
 - Rec_Data 在哪一個 Bank ?
 - 組合語言中，banksel Rec_Data 會被組譯成什麼指令?

Ex7-3

- 將練習 **Ex6-1a** 的 **Ex6-1a.c** 程式拆成二個 **C** 程式
 - Ex7-3.c : 主程式
 - Ex7-3a.c : Initial 函數集
- 在 **Project** 中加入 **Ex7-3a.c**

重點提示:

1. 在 **C** 底下，函數及公用變數已經是 **GLOBAL**
2. **C** 程式要呼叫外部 **C** 的函數時，一定要有外部函數的“函數的原型宣告”
3. 外部函數所宣告的變數，如要在本程式下使用，可用 **extern** 的宣告



MICROCHIP

用組合語言完成標準C函數

**Using Assembly for
C Standard Function Call**

參數的傳遞

C 呼叫組合語言函數

- **C**傳入的參數是被放置在軟體堆疊裡
 - FSR1為堆疊指標，FSR2為框架指標
 - 回傳值一般是放在 PRODL:PRODH (16bit)
- 參數的堆疊排列順序是從右到左
 - strcpy(char *dest, char *src)
 - 指標變數src首先被推入軟體堆疊
 - 接著指標變數dest則會被放在下一個軟體堆疊
- 堆疊位址的設定是在**18F4520.lkr**中
 - STACK SIZE=0x100 RAM=gpr4
 - 堆疊最底層為0x400,其次為0x401,0x402.....
 - 不同的 PIC 因 RAM 容量不同，軟體堆疊設定不盡相同

組合語言傳回參數

傳回值的大小 傳回值放置的地方

8 bit

WREG

16 bit

PRODH:PRODL

24 bit

AAREG2+2:AAREG2+1:AAREG2

32 bit

AREG3+3:AAREG3+2

:AAREG3+1:AAREG3

>32 bit

In the Caller's Stack Frame

C呼叫組合語言的準備

Example - strlen()

傳入的參數存入堆疊

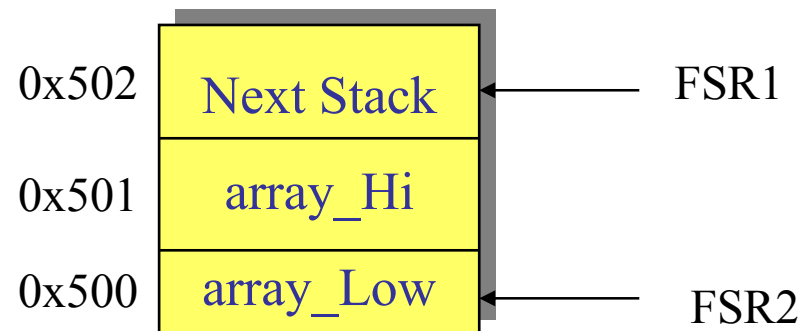
```
char array[10]="12345";
#pragma udata abc=0x123
char D_Len;
#pragma udata

void main(void)
{
    D_Len=strlen(array);
}
```

編譯後變數位址的安排如下：

```
array  --> 0x0080
D_Len  --> 0x0123
FSR1   --> 0x0500
FSR2   --> 0x0500
```

```
Movlw    low(array)      ;0x80 --> Wreg
movwf    POSTINC1        ;FSR1=0x501
movlw    hi(array)       ;0x00 --> Wreg
movwf    POSTINC1        ;FSR1=0x502
call     strlen
movf     POSTDEC1,F       ;FSR1=0x501
movf     POSTDEC1,F       ;FSR1=0x500
MOVff    WREG,D_Len       ;Return Length
```



被呼叫的組合語言函數

Example - strlen()

```

Strlen    MOVLW    -2          ; 0xFE
           MOVFF    PLUSW1,FSR0L
           MOVLW    -1          ; 0xFF
           MOVFF    PLUSW1,FSR0H
           CLRF     PRODL
           CLRF     PRODH
jLoop     MOVF     POSTINC0
           BZ       jEnd
           INFSNZ   PRODL
           INCF     PRODH
           BRA      jLoop
jEnd      MOVF     PRODL,W
           RETURN
    
```

- 自**FSR1**指到的堆疊位址取出參數(此處為指標)傳給**FSR0**
- 堆疊運算採 **2'S**
 - $-2 = \text{FSR1} + 0xFFE$
 - $-1 = \text{FSR1} + 0xFFF$
- **FSR1**永遠指向堆疊最頂端
- 如在函數內需使用到**FSR2**，則必須將**FSR2**存到**FSR1**指到的堆疊，不可任意存放在**RAM**區
 - Auto 變數必須放在堆疊裡
- 傳回值會放在**PRODX**

較複雜的參數傳遞

Example - strcpy()

```
#pragma udata abc=0x123
const char a[10]= {"12345678"};
char b[10];
char *c;
#pragma udata

void main(void)
{
    Nop();
    c = strcpy ( b , a );
    while(1);
}
```

```
movlw    low(a)           ; array a low point
movwf    POSTINC1         ; pass to (stack+0)
movlw    hi(a)            ; array a hi point
movwf    POSTINC1         ; pass to (stack+1)
;
movlw    low(b)           ; array b low point
movwf    POSTINC1         ; pass to (stack+2)
movlw    hi(b)            ; array b hi point
movwf    POSTINC1         ; pass to (stack+3)
;
callstrcpy
;
movff    PRODH,c+1        ; save return to c
movff    PRODL,c
movlw    0x04              ; Stack-4
subwf    FSR1L,F
```

較複雜的參數接收

Example - strcpy()

因本函數有使用到FSR2
故須將FSR2推入堆疊中

```

STRING      CODE
global strcpy

;
strcpy      movff    FSR2L,POSTINC1      ; Save FSR2 on the stack.
            movff    FSR2H,POSTINC1
;
            movlw    -6                  ; Load FSR2 with the 'src' pointer
            movff    PLUSW1,FSR2L
            movlw    -5
            movff    PLUSW1,FSR2H
;
            movlw    -4                  ; Load FSR0 with the 'dest' pointer
            movff    PLUSW1,FSR0L
            movlw    -3
            movff    PLUSW1,FSR0H
;
            movff    FSR0L, PRODL        ; make a copy for the return value
            movff    FSR0H, PRODH
;
jLoop:      movff    POSTINC2, INDF0      ; Copy source to destination
            tstfsz   POSTINC0,ACCESS     ; Test 'dest' for '\0'
            bra      jLoop
;
            movlw    0xFF                ; restore FSR2
            movff    PLUSW1,FSR2H
            movf     POSTDEC1,F,ACCESS   ; decrement FSR1
            movlw    0xFF
            movff    PLUSW1,FSR2L
            movf     POSTDEC1,F,ACCESS   ; decrement FSR1
            return

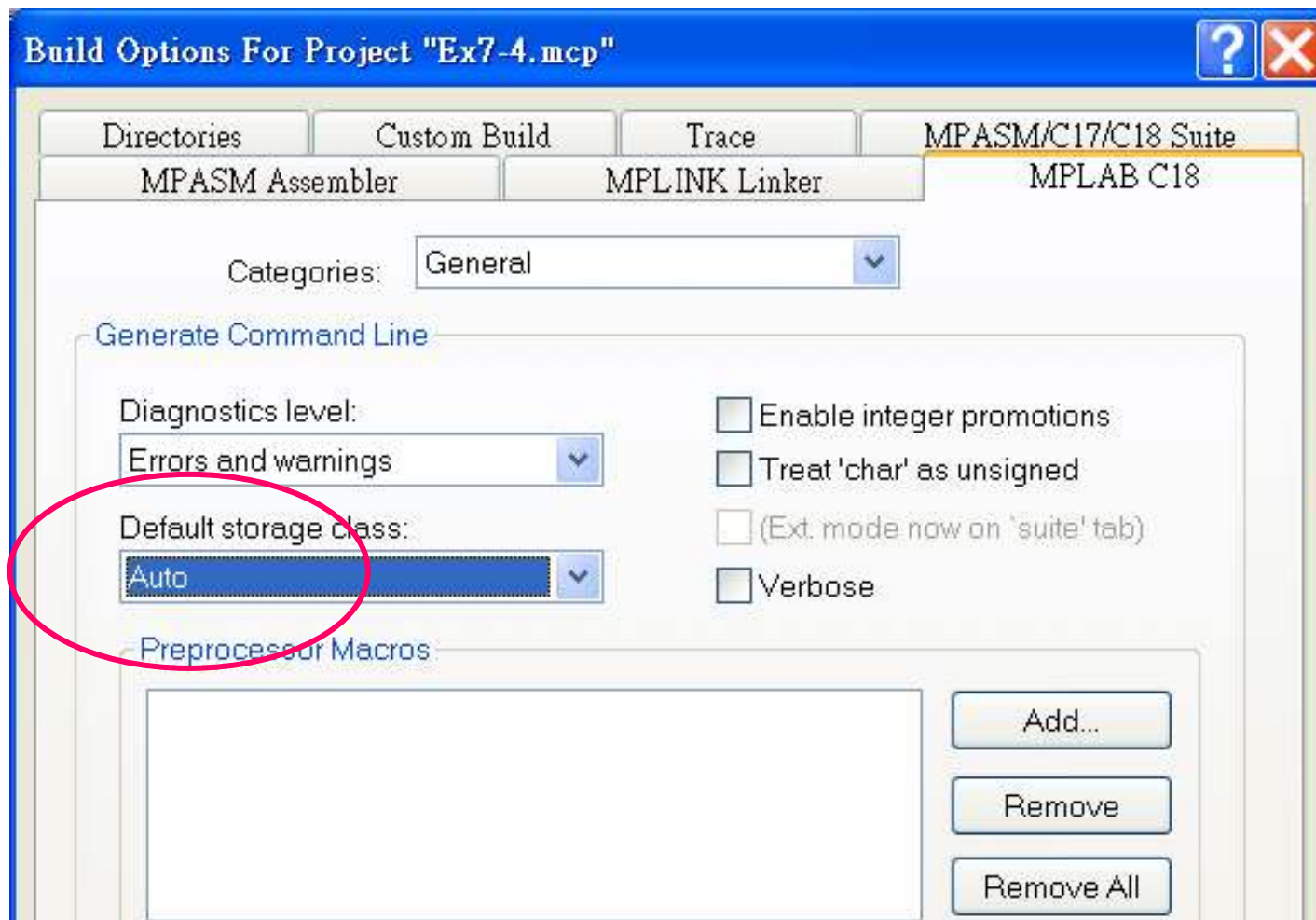
```

堆疊內容

0x506	未使用	0
0x505	FSR2H	-1
0x504	FSR2L	-2
0x503	Hi b	-3
0x502	Low b	-4
0x501	Hi a	-5
0x500	Low a	-6

FSR1 Offset

參數傳遞選擇 (RAM or Stack)



Ex7-4

使用參數傳遞的組合語言函數

- 將練習 6-1的程式中的 **putrsUSART()** 函數改用組合語言撰寫(傳入**ROM** 指標)
 - 在組合語言宣告: `global Send_ROM_USART`
 - 在Ex7-4.c作原型宣告:
 - `void Send_ROM_USART(const rom char *data)`
 - 將顯示的 `Disp_Msg[]` 改為 `const rom char`
 - 傳入之 **ROM** 指標為 `near` ,只有16 bit
- 在 **Project** 中加入 **S_R_UART.asm**
 - 功能與 `putrsUSART()` 相同

課程結束！



MICROCHIP

感謝您對 **Microchip** 的支持

如有任何問題請電

0800-717718

或

www.microchip.com.tw