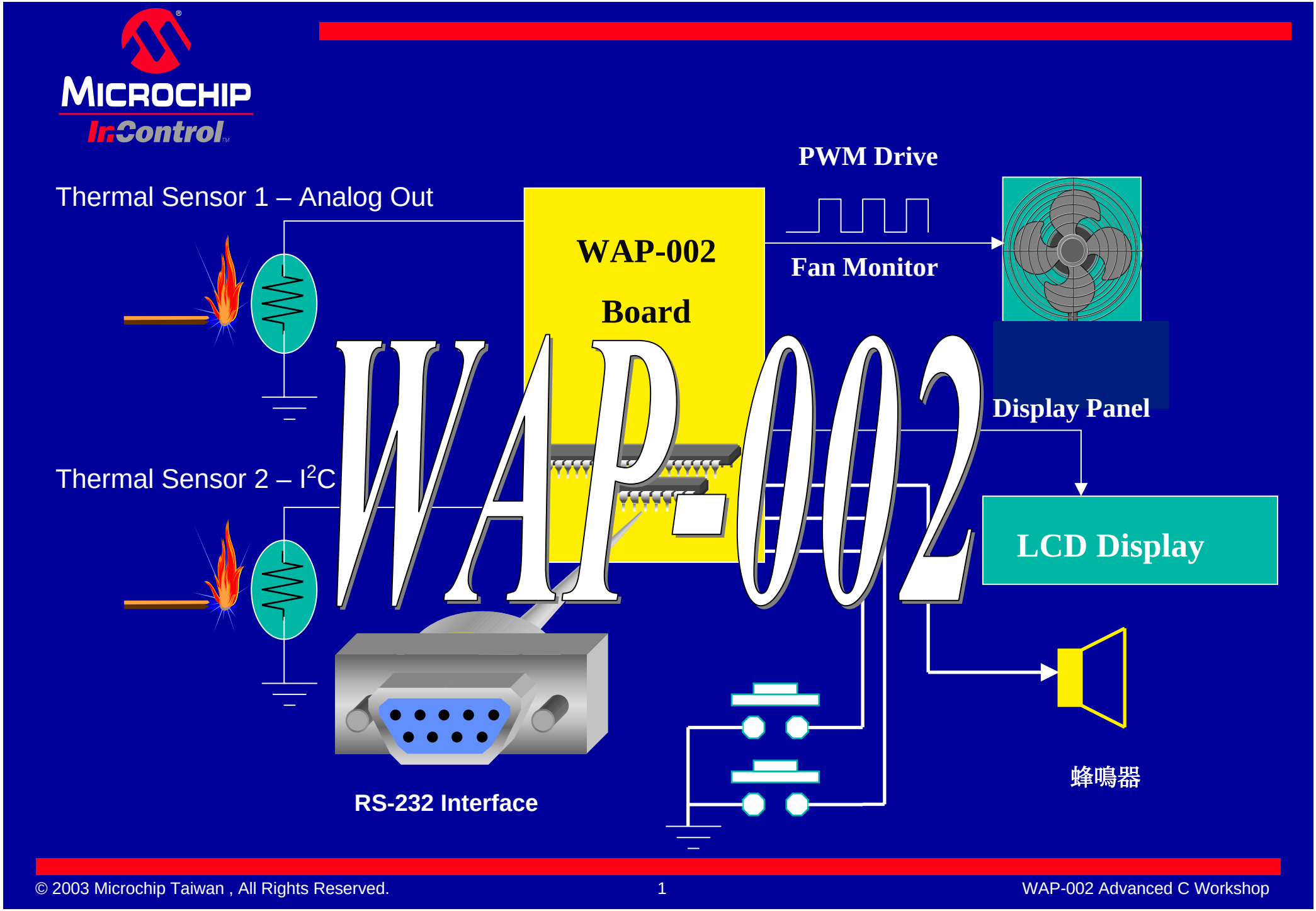


歡迎參加 WAP-002 教育訓練

- 使用軟體工具
  - MPLAB IDE v6.10.00 (或更新版本)
  - MPASM , MPLINK , MPLIB
  - MPLAB C18 v2.10.00 (或更新版本)
- 使用硬體工具
  - MPLAB ICD2
  - Microchip APP001 Workshop Board (PIC18F452 inside)
- 參考書籍
  - MPASM User's Guide with MPLINK and MPLIB (DS33014)
  - MPLAB IDE v6.10 中文使用手冊
  - PIC18Fxx2 Data Sheet (DS39564A)

WAP002 課程介紹（一）

- 第零章
  - MPLAB 發展工具的環境
- 第一章
  - MPLAB C18 重點複習
- 第二章
  - LCD 顯示模組的控制
  - 練習一 ： 驅動 LCD 顯示模組
- 第三章
  - 溫度的量測與顯示
  - 練習二：將測到的兩種溫度顯示到 LCD

WAP002 課程介紹（二）

- 第四章
  - 使用 VT-100 終端機
  - 練習三：顯示溫度到 VT100 的固定位置
- 第五章
  - 存取內部 EEPROM
  - 練習四：自 VT100 輸入設定溫度並存入 EEPROM
- 第六章
  - PWM 輸出計算
  - 練習五：計算出目前溫度的 PWM 輸出值
- 第七章
  - 完成一智慧型溫度控制警報系統
  - 練習六：高、低溫控制顯示與警報音

第零章

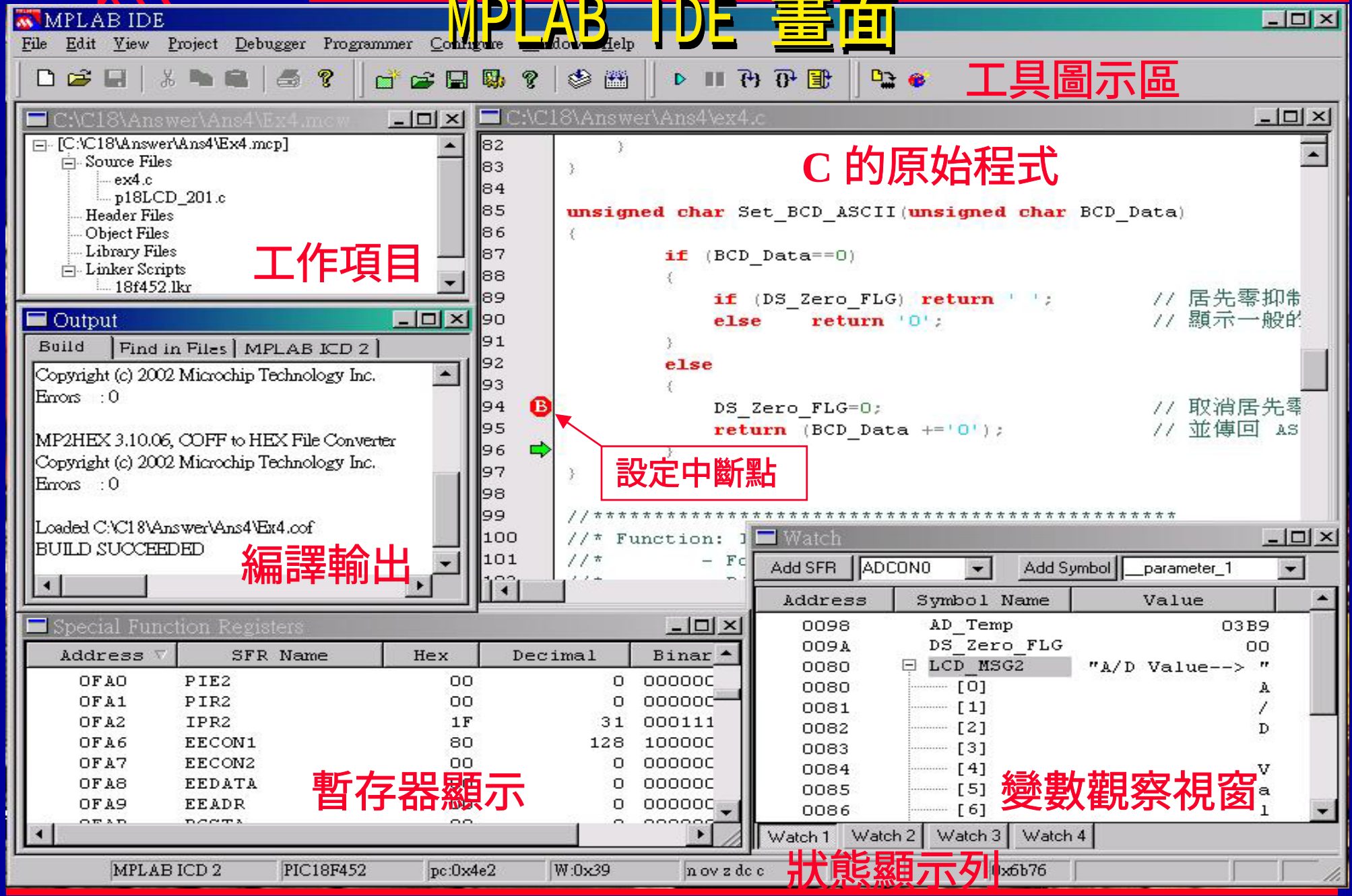
1. MPLAB IDE v6.xx
2. MPLAB ICD2
3. APP001 實驗板

MPLAB-IDE 功能介紹 (v6.xx)

- 高整合度的微處理器軟體 / 硬體研發及偵錯平台
- 採用專案管理模式 (Project)
- 多視窗原始程式編輯、修改
- 直接組譯 / 編譯原始程式
- 軟體模擬
- 具有輸入模擬功能
- 支援原始程式偵錯
- 支援 MPASM、 MPLINK
- 支援 C compiler

- 硬體除錯工具：
  - MPLAB-ICE 4000
  - MPLAB-ICE 2000
  - MPLAB-ICD2
- 程式燒錄工具：
  - PRO MATE - II
  - PICSTART Plus
  - MPLAB-ICD2

MPLAB IDE 畫面



MPLAB IDE 硬體支援

硬體設備	MPLAB v6.xx	MPLAB v5.xx
MPLAB-ICE4000	Yes	No
MPLAB-ICE2000	Yes	Yes
PICMASTER	No	Yes
ICEPIC	No	Yes
MPLAB-ICD	No	Yes
MPLAB-ICD2	Yes	Yes
PROMATE	Yes	Yes
PROMATE-II	Yes	Yes
PICSTART Plus	Yes	Yes

MPLAB™ v6.xx
整合式的發展環境

內含多功能程式編輯器

原始檔案程式偵錯功能

單一系統專案管理模式

語言工具	軟體模擬	硬體模擬器	程式燒錄器
MPASM 編譯器	MPLAB-SIM 軟體模擬器	MPLAB-ICE 2000 4000	PICSTART® Plus
MPLINK 連結器	dsPIC 軟體模擬器	PICMASTER (V5.xx)	PRO MATE®
MPLAB Cxx dsPIC C 編譯器		MPLAB ICD2	

MPLAB ICD2

- **Windows 98 SE / 2000 / XP**
- **支援 USB 及 RS - 232 介面**

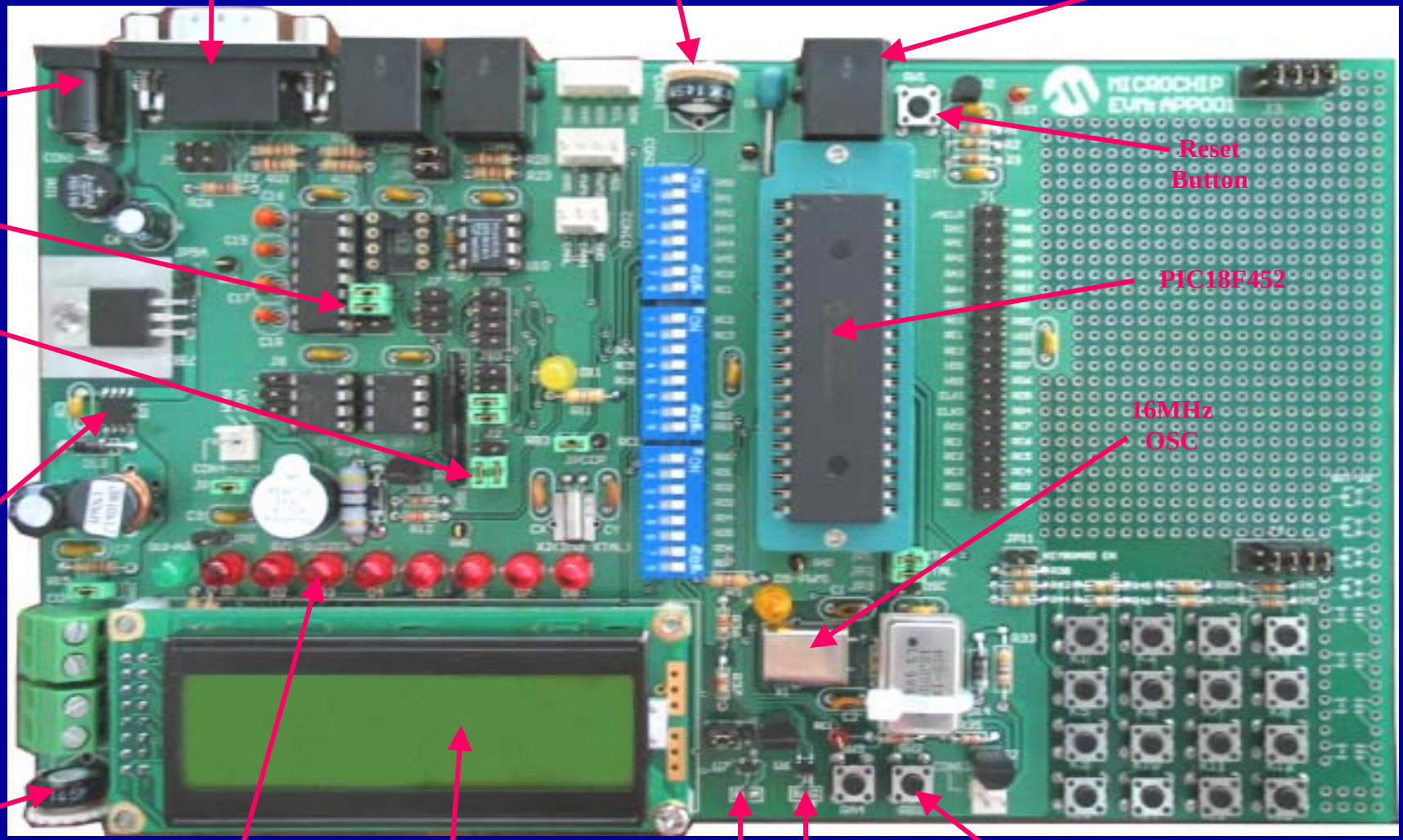
MPLAB-ICD2 功能

- 全速執行
- 單步執行
- 單點硬體中斷與 Pass Count 設定
- 變數觀察，原始程式除錯等級
- 快速載入程式到模擬元件
- 可當模擬元件的燒錄工具
- 工作電壓 ： 2.5V to 5.5V
- 頻率範圍 ； 32KHz to 20MHz
- RS-232 或 USB 介面
- 價格便宜
- 直接使用在 MPLAB-IDE v6.xx

MPLAB-ICD2 注意事項
PIC18F452除錯時

- ICD2 會佔用 18F452 最後的程式空間 (除錯程式使用)
- 任何修改程式的動作，需重新執行燒錄動作
- 程式記憶體及 EEPROM 的內容值，在除錯的過程中不會自動更新，如需更新需以讀取 Device 的方式更新
- ICD2 不支援堆疊視窗功能
- ICD2 不支援 SLEEP 功能
- ICD2 在除錯的過程中不能啟動 Watch-Dog Timer
- ICD2 再執行燒錄功能時，EEPROM 的內容會被清除
- RB6 & RB7 保留給 ICD 做除錯用
- MCLR pin 會出現 13V 的電壓 (thru a 1kohm resistor)

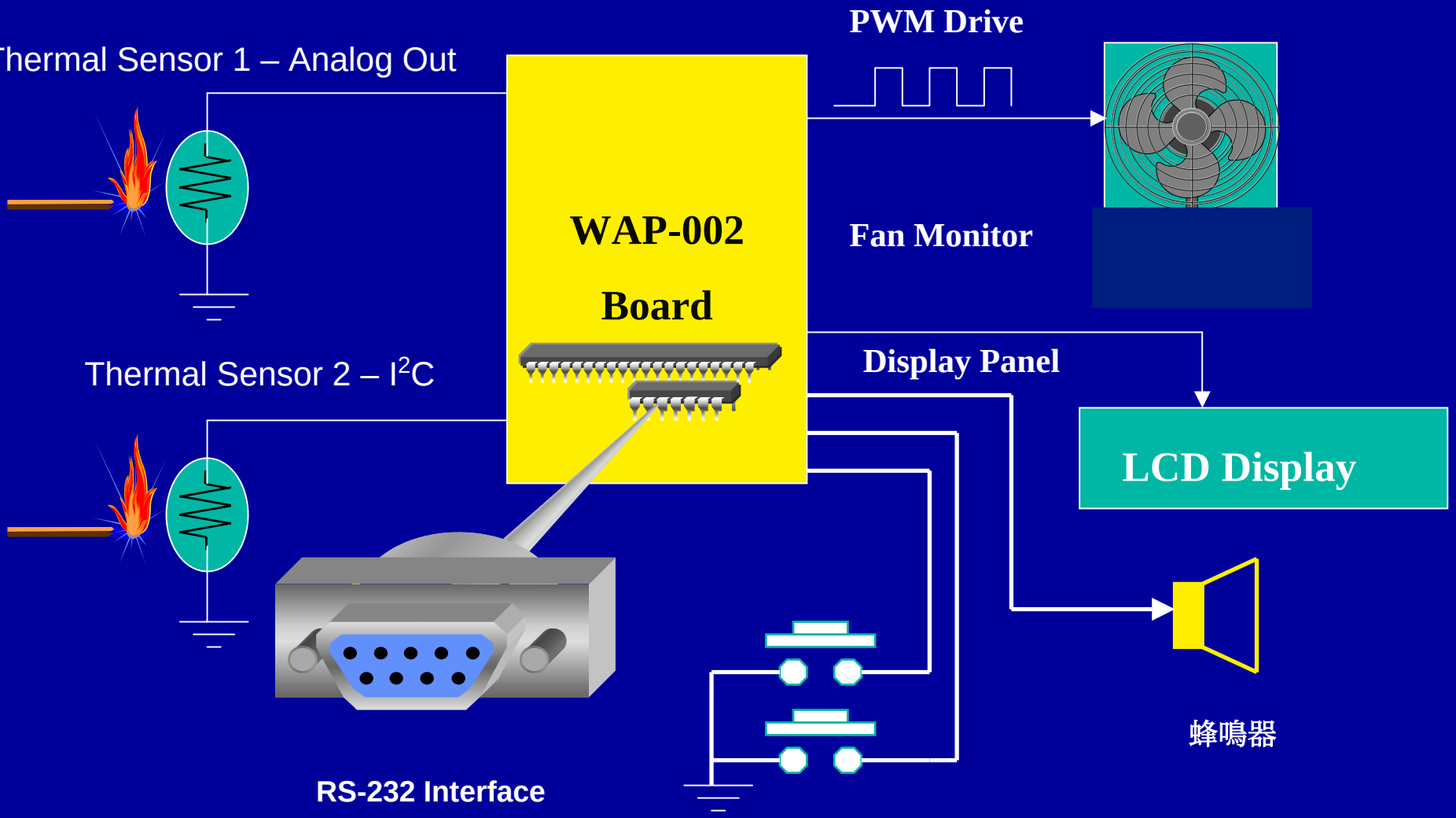
APP001 實驗板



APP001 實驗板功能

- 內建兩個溫度感應器，標準參考電壓源，10-bit A/D 轉換器
  - I²C 溫度感應器(TC74A)，線性輸出溫度感應器(TC1047A)
  - 4.096V 參考電壓 (MCP1541)
- 串列通訊介面
  - SPI，I²C，RS-232，RS-485，CAN，ICD
- 其它周邊元件
  - 2x16 LCD Module，8 個顯示的 LED，PWM 亮度顯示 LED
  - I²C EEPROM，SPI EEPROM，內建式 EEPROM
  - 兩個按鍵開關，PWM風扇驅動，蜂鳴器
- 振盪電路
  - 16MHz / 32768Hz 石英晶體，4MHz 振盪器

WAP-002 智慧型警報系統



WAP-002 智慧型警報系統需求

- 量測兩個溫度感應器，並顯示目前溫度在 LCD 及 VT-100 終端機
- User 可透過 VT-100 終端機的鍵盤來設定最高、最低溫度的控制點並存入 溫度控制站的 EEPROM
- User 可透過 VT-100 終端機開啟或關閉溫度控制站的控制模式
- 溫度警報系統的輸出為一個 10-bit PWM 輸出，其輸出 (Duty) 會隨 EEPROM 的最高、最低溫度設定點與目前所測得的溫度計算出 Duty 的輸出值 (1% to 99%)，以控制散熱風扇的轉速
- 溫度警報系統有一蜂鳴器，會隨溫度發出不同的警報聲響
  - 溫度超過 98% -- 連續警報音
  - 溫度介於兩點間的 75% - 97% -- 段續警報音
  - 溫度介於兩點間的 1% - 74% -- 關閉警報音
  - 溫度低 1% -- 段續長聲警報音

第一章

MPLAB C18 重點複習

Options

1. 資料與變數型態
2. 變數與程式位址的安排
3. 中斷處理
4. C18 的啟動模組

資料與變數型態

C 的資料型別

C 的變數類別

資料型別

MPLAB C18 提供的各項資料型別

型 別	字 元 數 (Bits)	數 字 範 圍 (Range)
void	N/A	N/A
char	8	-128 ~ +127
unsigned char	8	0 ~ 255
int (或 short)	16	- 32,768 ~ 32,767
unsigned int	16	0 ~ 65,535
short long	24	- 8,388,608 ~ 8.388.607
unsigned short long	24	0 ~ 16,777,215
long	32	- 2,147,483,648 ~ 2,147,483,647
unsigned long	32	0 ~ 4,294,967,295
float	32	1.7549435e-38 ~ 6.80564693e+38
double	32	1.7549435e-38 ~ 6.80564693e+38

資料儲存的順序

- 使用 "*較小值使用低位址*" 格式
- 若一個長整數使用以下的方式宣告：

```
#pragma idata test=0x0200
long Var = 0xAABBCCDD ;
```

  - 則 Var 變數值存於記憶體的實際情形如下：

RAM Address	0x200	0x201	0x202	0x203
Content	0xDD	0xCC	0xBB	0xAA

- 使用 MPLAB IDE 的 "Watch" 功能可驗證這樣的安排順序

記憶體模式 (near , far & rom , ram)

	程式記憶區(rom)	資料記憶區(ram)
far	2M bytes 定址模式 (用24-bit的指標)	4K bytes 定址模式 (用16-bit的指標)
near	<64K bytes定址模式 (用16-bit的指標)	Access RAM 定址模式 (用8-bit的指標)

注意：紅色字框為內定的記憶體模式 (Default)

記憶體模式 – 範例說明

```
const rom far char LCD_MSG1[ ]="PIC18F452";
```

在16-Bit定址中擴展為24-bit位址存取

指定用程式記憶體

常數宣告 (不可變更)

```
int AD_Read;
```

宣告變數在 4K RAM的區域

宣告變數在 Access RAM

```
near int AD_Read;
```

變數的類別（一）

區域變數（ local ，auto ）

- 區域變數是在函數內部所宣告的變數，其視野僅在本函數內，其它的函數無法使用。
- 區域變數的名稱在不同的函數內可相同。
- 區域變數的生命週期是函數被使用時開始存在，函數執行結束時消失。

```
Void main(void)
{
    unsigned char i=0;
    i++;

    func();
}

/* function call */
void func(void)
{
    unsigned char i=0;

    i--;
}
```

變數的類別（二）
公用變數（Global）

- 公用變數是在函數以外的地方宣告，實際佔有記憶體的變數
- 只要是視野內的函數均可存取此變數
- 若要在程式中使用其它檔案所宣告的公用變數時，可使用extern來達成
- 公用變數的生命週期永遠存在

```
extern unsigned char KeyData
unsigned int ADResult;

void Motor(void)
{
    if (KeyData == 0x80)
    {
        TRISCbits.TRISC2=0;
        ConvertADC();
        while(BusyADC());
        ADResult= ReadADC();
        SetDCPWM1(ADResult);
    }
    else
        TRISCbits.TRISC2=1 ;
}
```

變數的類別（三）
靜態變數（Static）

- 靜態變數的視野與區域變數是一樣的，只能在函數內部；但生命週期並不會隨函數執行結束而消失(保留下來)
- 靜態變數的值存在於固定的記憶位址
- 當該函數再次執行時，上次保留之變數值會繼續使用

```
Void main (void)
{
    unsigned char i;

    for (i;i<10;i++)
    AddOne( );
}

/*** AddOne Function ***/
void AddOne(void)

{
  static unsigned char count=0;
  count++;
}
```

變數的類別（五）
揮發性變數（Volatile）

- 變數的值不一定要經由程式來改變，變數本身會自行或隨外在因素而改變
- 揮發性變數一般就是指某些特殊暫存器(詳細請參考p18f452.h)：
  - TMR0 , TMR1 ...
  - PC , PCL ...
  - EEDATA , ADCON0 ...
  - PORTA , PORTB ...

```
Unsigned char x,y,;
volatile unsigned char TMR0;

x=55;
y=x;

TMR0=0x00;
/*------------------------------
    The compiler must read TMR0
    and can't use the 0x00 in its
    temporary variable since TMR0
    increments with execution
------------------------------*/
y=TMR0;
```

變數的類別（六）
自定型別(Typedef)

- Typedef 用來創造自己的型別名稱
- 主要的目的：
  - 提高程式的攜帶性
  - 讓程式更容易閱讀
  - 減少原始程式碼

```
typedef unsigned char Byte;
typedef unsigned int Word;

void main(void)
{
    Word j[10];
    Byte i;

    for (i=0;i<10;i++)
        j[i] = (Word)i;
}
```

函數的原型宣告 - Prototype

- 函數的原型宣告是讓編譯器檢驗函數所使用的參數型態
- User自訂的函數均需宣告
- 在多個原始程式中，最好將原型宣告整理成一個 xxx.h 的檔案讓每個程式連結進來
- 在 A 程式宣告的函數原型，B 程式是看不到的 ???
  - B 程式無法正確呼叫 A 程式的函數，並且會在 link 時產生錯誤
  - 解決方法：將 A 程式的函數原型也在 B 程式也做宣告
- 善用 #include，可以讓數個程式輕易的使用既有的函數
  - 例如 : #include <timers.h>

變數與程式位址的安排

變數位址安排

程式位址安排

C18 變數位址安排

- MPLINK 自動安排變數位址
- 安排變數於特定的位址
  - 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/idata

- **#pragma udata [data-qualifier] [section-name [= addr]]**
  - **`#pragma udata section-name`** 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]`**：變數放置的區域
    - "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 : 按照 Linker 的連結描述檔內的 section-name, 進行指定變數位址安排
    - 不指定section-name : Linker自動安排， 變數放在 unprotected 區
    - = addr :強制設定該 section-name 的變數位址

#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;

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

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

#pragma udata
```

宣告以下之變數放在Access Bank中，由Linker自行安排位址

宣告以下之變數放在GPR中位址為0x100的地方

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

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

C18 ROM Data 位址安排

- 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

C18 程式位址安排

- 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.lkr)
  - 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
```

C18 Configuration Words

- 指定 Configuration Words 的預設值於程式中
  - PIC18FXXX 的 Config. Words 位於程式記憶體位址 0x300000 – 30000D
  - 這些位址中相關的位元必須被正確地燒錄至 IC 中才可使程式正常運作

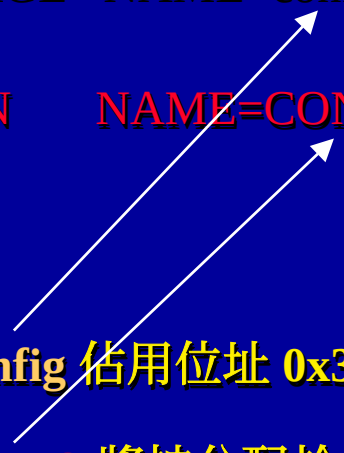
File Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300001h	CONFIG1H	—	—	OSCSEN	—	—	FOSC2	FOSC1	FOSC0	--1- -111
300002h	CONFIG2L	—	—	—	—	BORV1	BORV0	BODEN	PWRTEN	---- 1111
300003h	CONFIG2H	—	—	—	—	WDTPS2	WDTPS1	WDTPS0	WDTEN	---- 1111
300005h	CONFIG3H	—	—	—	—	—	—	—	CCP2MX	---- ---1
300006h	CONFIG4L	$\overline{\text{DEBUG}}$	—	—	—	—	LVP	—	STVREN	1--- -1-1
300008h	CONFIG5L	—	—	—	—	CP3	CP2	CP1	CP0	---- 1111
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000Ah	CONFIG6L	—	—	—	—	WRT3	WRT2	WRT1	WRT0	---- 1111
30000Bh	CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
30000Ch	CONFIG7L	—	—	—	—	EBTR3	EBTR2	EBTR1	EBTR0	---- 1111
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEh	DEVID1	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	(1)
3FFFFFh	DEVID2	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0000 0100

C18 Config. Words 的設定

- 讓程式直接設定好 Config. Words 以減少設定錯誤
- 在程式中最好能將 Configuration Words 都定義清楚
  - 有助於日後程式的維護
  - **Config. Bits 的位址定義在 PIC18F452.lkr**

```
CODEPAGE   NAME=config   START=0x300000   END=0x30000D   PROTECTED

SECTION    NAME=CONFIG    ROM=config
```



  - **程式頁 config 佔用位址 0x300000 to 0x30000D 並且是被保護的**
  - **節區 CONFIG 將被分配於 Program Memory 中，且使用 config 程式頁佔用的區間**

C18 Config. Bits 設定範例

- 指定 Configuration Words 的預設值於程式中(cont.)
  - #pragma romdata CONFIG 來指定此一特定的位址的設定值
  - CONFIG 為 PIC18F452.lkr 中已經宣告的節區

```
#pragma romdata CONFIG
const rom unsigned char CONFIG1L=0xff ;                 // Don't care byte
const rom unsigned char CONFIG1H=0b00100010 ;           // Disable OSC switch , XXXXX010 = HS Osc
const rom unsigned char CONFIG2L=0b00000001 ;           // Disable PWRT , Disable BOR
const rom unsigned char CONFIG2H=0b00000000 ;           // Disable WDT timer
const rom unsigned char CONFIG3L=0xff ;                 // Don't care byte
const rom unsigned char CONFIG3H=0b00000000 ;           // XXXXXXX0 = CCP2 --> RB3
const rom unsigned char CONFIG4L=0b00000001 ;           // 0XXXXXXX Backgroung Debug Enable ( ICD )
const rom unsigned char CONFIG4H=0xff ;                 // Don't care byte
const rom unsigned char CONFIG5L=0b00001111 ;           // Not PROG code protected
const rom unsigned char CONFIG5H=0b11000000 ;           // Not EEPROM code protected
const rom unsigned char CONFIG6L=0b00001111 ;
const rom unsigned char CONFIG6H=0b11100000 ;
const rom unsigned char CONFIG7L=0b00001111 ;
const rom unsigned char CONFIG7H=0b01000000 ;
#pragma romdata
```

中斷處理

18F452 中斷處理

- 18F452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，RB0 除外）
  - 每個中斷源均有獨立的中斷旗標(Flag)
  - 每個中斷源均可Enable或Disable
  - 中斷旗標的清除==>自行用軟體清除
    - USART 產生的 TXIF 及 RCIF 旗標， 無法直接用軟體清除 Fla
    - 清除 TXIF → 寫入 TXREG
    - 清除 RCIF → 讀取 RCREG

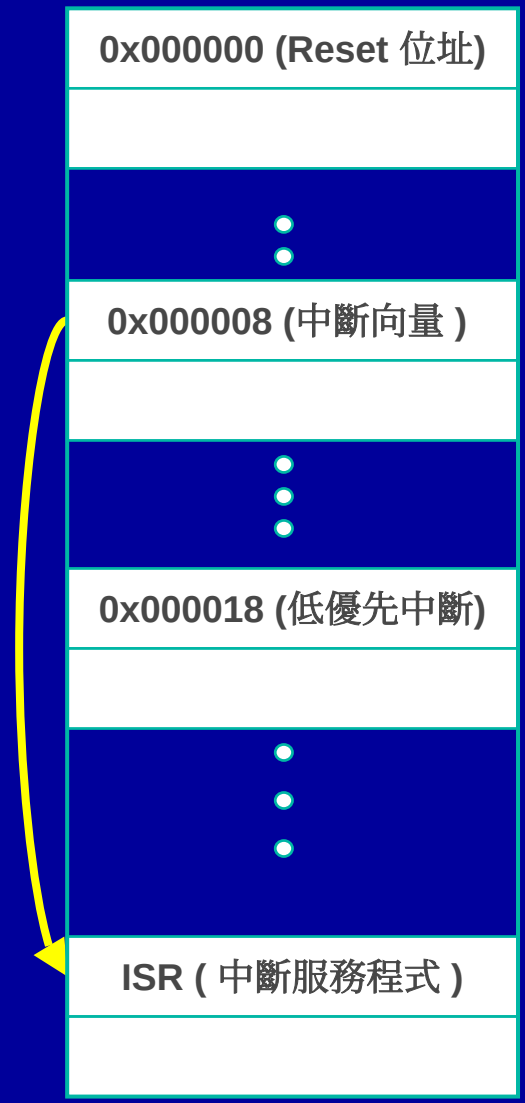
Shadow 暫存器

- Shadow Register
  - 提高中斷程式對事件的反應速度
- 高優先權中斷
  - W，BSR，STATUS 自動存入Shadow Register
  - RETFIE FAST：自 Shadow Register 取回暫存值
- 低優先權中斷
  - W，BSR，STATUS 的存入、取出需透過軟體堆疊
  - 程式的返回：RETFIE 0

將控制權轉移給 ISR

- PIC18F 的中斷向量分別為 0x000008 及 0x000018
  - 兩向量之間只相距 16 個 bytes 的空間
  - 在向量位址上, 使用 goto 將程式的執行交給指定的 ISR (Interrupt Service Routine)

```
#pragma code hi_vector=0x0008   // 設定中斷進入點
void isr_high_code(void)
{
    _asm
    goto        isr_high
    _endasm
}
#pragma code
```



設定高優先權中斷服務程式

- 利用前置處理指令“#pragma interrupt”來指明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
}
#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
```

使用內建組合語言功能，轉移控制權到中斷服務程式(isr_high)

中斷服務程式(isr_high)

設定低優先權中斷服務程式

- “#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
- “#pragma interruptlow”來指明該函數為低優先權中斷服務程式
- W, STATUS, BSR 的存取用軟體堆疊來達成，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)

C18 的三個幫手

- 微控制器的標準名稱定義檔
  - p18f452.h ， p18f8720.h
- 微控制器的周邊位址設定檔
  - p18f452.lib， p18f448.lib
- C18 的起動模組
  - Reset Vector 的控制權
  - 初始變數如何處理

MPLAB-C18 啟動模組

- 啟動模組的功能(一定要用)
  - 掌管最初的執行、規劃 (Power-On Reset)
  - 規劃軟體堆疊區
  - 設定變數初始值
  - 轉移控制權到 main( ) 函數
- 啟動模組有三個，放在 clib.lib 函數庫裡
  - c018.o – 無須規劃初始變數時使用
  - c018i.o – 須規劃初始變數時使用
  - c018iz.o – 先將RAM清除，再規劃初始變數值
  - 原始程式位置：\mcc18\src\startup

C018i.c 啟動模組程式

```
#pragma code _entry_scn = 0x000000
static void
entry (void)
{ _asm   goto _startup  _endasm  }

#pragma code _startup_scn
static void _startup (void)
{
  _asm
    // Initialize the stack pointer
    LFSR 1, _stack LFSR 2, _stack CLRF TBLPTRU, 0
   // Initialize rounding flag for floating point libs
    BSF  FPFLAGS,RND,0
   _endasm

  _do_cinit ( );

loop:
  // Call the user's main routine
  main ( );
  goto loop;
}                          /* end _startup() */
```

RESET 位址: 0x000000

給予 STACK 及 TBLPTRU 初使值

設定初始變數值

將控制權交至 main( )

在 18F452.lkr 描述檔

```
// Sample linker command file for 18F452
// $Id: 18f452.lkr,v 1.3 2002/07/29 19:09:08 sealep Exp $

LIBPATH  .

FILES c018i.o
FILES clib.lib
FILES p18f452.lib

CODEPAGE   NAME=vectors  START=0x0        END=0x29       PROTECTED
CODEPAGE   NAME=page     START=0x2A       END=0x7FFF
CODEPAGE   NAME=idlocs   START=0x200000   END=0x200007   PROTECTED
CODEPAGE   NAME=config   START=0x300000   END=0x30000D   PROTECTED
CODEPAGE   NAME=devid    START=0x3FFFFE   END=0x3FFFFF   PROTECTED
CODEPAGE   NAME=eedata   START=0xF00000   END=0xF000FF   PROTECTED

ACCESSBANK  NAME=accessram  START=0x0      END=0x7F
DATABANK    NAME=gpr0       START=0x80     END=0xFF
DATABANK    NAME=gpr1       START=0x100    END=0x1FF
DATABANK    NAME=gpr2       START=0x200    END=0x2FF
DATABANK    NAME=gpr3       START=0x300    END=0x3FF
DATABANK    NAME=gpr4       START=0x400    END=0x4FF
DATABANK    NAME=gpr5       START=0x500    END=0x5FF
ACCESSBANK  NAME=accesssfr  START=0xF80    END=0xFFF     PROTECTED

STACK SIZE=0x100 RAM=gpr5
```

連結起動模組

連結標準函數庫

連結18F452周邊函數庫

特殊的巨集指令

下列的 PICmicro的巨集指令可直接在C18裡直接執行

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<--B7<--B6…….B0<--C)
Rlncf(Var)	將Var向左旋轉(不含進位旗號) (B7<--B6…….B0<--0)
Rrcf(Var)	將Var與進位旗號一起向右旋轉 (C-->B7-->B6……B1-->B0-->C)
Rrncf(Var)	將Var向右旋轉(不含進位旗號) (0-->B7-->B6……B1-->B0)
Swapf(Var)	將Var高、低4位元互換
說明: Var必須是一個8位元的標準變數，且不可被存放在堆疊中	

第二章
LCD 模組控制

硬體設計

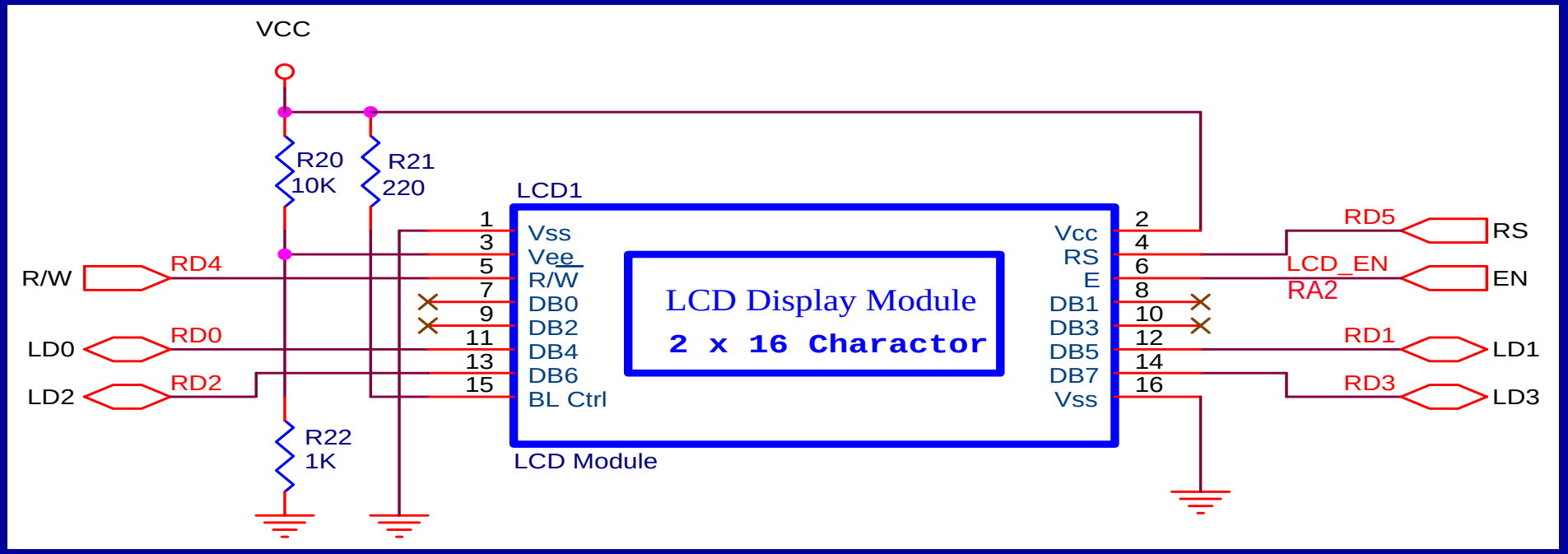
HD44780 控制指令

4-bit 控制方式

HD44780-Based LCD 模組

Pin number	Symbol	Level	I/O	Function
1	Vss	-	-	Power supply (GND)
2	Vcc	-	-	Power supply (+5V)
3	Vee	-	-	Contrast adjust
4	RS	0/1	I	0 = Instruction input 1 = Data input
5	R/W	0/1	I	0 = Write to LCD module 1 = Read from LCD module
6	E	1, 1-->0	I	Enable signal
7	DB0	0/1	I/O	Data bus line 0 (LSB)
8	DB1	0/1	I/O	Data bus line 1
9	DB2	0/1	I/O	Data bus line 2
10	DB3	0/1	I/O	Data bus line 3
11	DB4	0/1	I/O	Data bus line 4
12	DB5	0/1	I/O	Data bus line 5
13	DB6	0/1	I/O	Data bus line 6
14	DB7	0/1	I/O	Data bus line 7 (MSB)

LCD 模組電路圖



電路的設計是採用**4-bit**介面

LCD 模組的接線

- LCD Module 使用的 I/O 接腳
  - PORTD RD0..RD3
    - 控制 LCD Module 的 DB4..DB7
  - PORTD RD4 → LCD Module 的 RS
    - 0 = command , 1 = data
  - PORTD RD5 → LCD Module 的 RW
    - 0 = write , 1 = read
  - PORTA RA2 → LCD Module 的 E
    - 1 = enable data r/w

- LCD Module 的使用模式
  - 4 Bits Mode , 2 Lines , 5X7 Character

HD44780 控制指令（一）

Instruction	Code										Description	Execution time**
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears display and returns cursor to the home position (address 0).	1.64mS
Cursor home	0	0	0	0	0	0	0	0	1	*	Returns cursor to home position (address 0). Also returns display being shifted to the original position. DDRAM contents remains unchanged.	1.64mS
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction (I/D), specifies to shift the display (S). These operations are performed during data read/write.	40uS
Display On/Off control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of all display (D), cursor On/Off (C) and blink of cursor position character (B).	40uS
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	*	*	Sets cursor-move or display-shift (S/C), shift direction (R/L). DDRAM contents remains unchanged.	40uS

HD44780 控制指令（二）

Cursor/display shift	0	0	0	0	0	1	S/C	R/L	*	*	Sets cursor-move or display-shift (S/C), shift direction (R/L). DDRAM contents remains unchanged.	40uS
Function set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length (DL), number of display line (N) and character font(F).	40uS
Set CGRAM address	0	0	0	1	CGRAM address						Sets the CGRAM address. CGRAM data is sent and received after this setting.	40uS
Set DDRAM address	0	0	1	DDRAM address							Sets the DDRAM address. DDRAM data is sent and received after this setting.	40uS
Read busy-flag and address counter	0	1	BF	CGRAM / DDRAM address							Reads Busy-flag (BF) indicating internal operation is being performed and reads CGRAM or DDRAM address counter contents (depending on previous instruction).	0uS
Write to CGRAM or DDRAM	1	0	write data								Writes data to CGRAM or DDRAM.	40uS
Read from CGRAM or DDRAM	1	1	read data								Reads data from CGRAM or DDRAM.	40uS

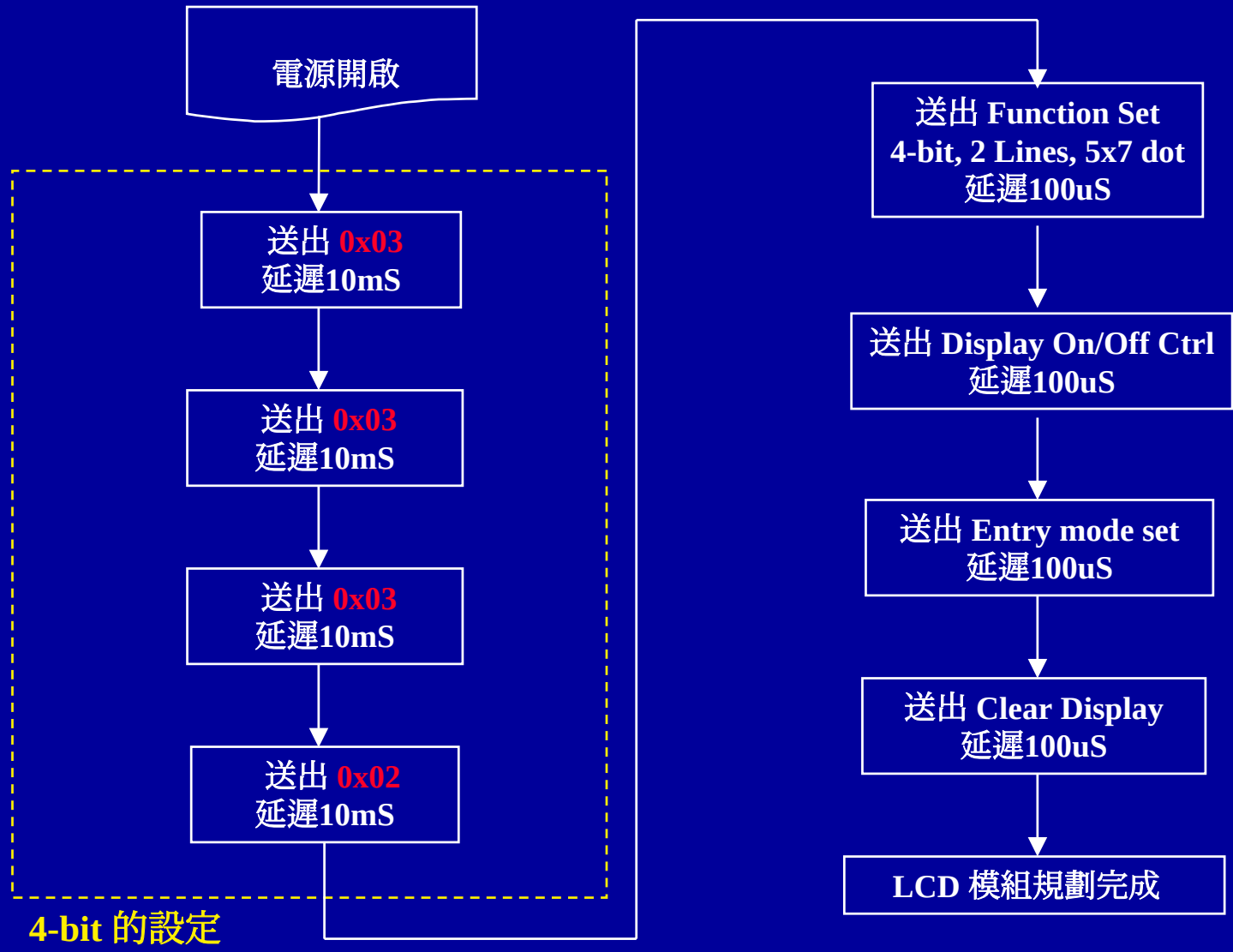
Remarks:

- DDRAM = Display Data RAM.
- CGRAM = Character Generator RAM.
- DDRAM address corresponds to cursor position.

HD44780 控制指令位元說明

Bit name	Settings	
I/D	0 = Decrement cursor position	1 = Increment cursor position
S	0 = No display shift	1 = Display shift
D	0 = Display off	1 = Display on
C	0 = Cursor off	1 = Cursor on
B	0 = Cursor blink off	1 = Cursor blink on
S/C	0 = Move cursor	1 = Shift display
R/L	0 = Shift left	1 = Shift right
DL	0 = 4-bit interface	1 = 8-bit interface
N	0 = 1/8 or 1/11 Duty (1 line)	1 = 1/16 Duty (2 lines)
F	0 = 5x7 dots	1 = 5x10 dots
BF	0 = Can accept instruction	1 = Internal operation in progress

4-bit 的初始設定



LCD 控制函數（一）

- OpenLCD(void)
  - 將 **LCD Module** 設定成 **4 Bits , 2 Lines , 5X7** 的操作模式
- WriteCmdLCD(unsigned char)
  - 寫入一個控制命令到 **LCD**
- WriteDataLCD(unsigned char) & putcLCD(unsigned char)
  - 將傳入的位元組以 **ASCII** 字元的型式顯示至 **LCD**
- LCD_Set_Cursor(unsigned char Y, unsigned char X)
  - 將 **LCD** 的游標重新設定於 **(Y,X)** 的位置上
- puthexLCD(unsigned char)
  - 將傳入的位元組以十六進制的型式顯示

LCD 控制函數（二）

- void putsLCD(unsigned char *Str)
  - 將指標 **Str** 所指到的 **Data Memory** 內的字串列顯示於 **LCD Module**
- void putrsLCD(unsigned char *Str)
  - 將指標 **Str** 所指到的 **Program Memory** 內的字串顯示於 **LCD Module**

請　注　意：

變數名稱前加一個 * 符號，　表示其為一個 **指標變數**；就是說它是存放指向某一個記憶體位址的指標

- 將指標變數加一，　實際上是將指標的位址指向下一個元素
  - 若指標被宣告為指向 char 的指標則其位址會被加 1
  - 若指標被宣告為指向 int 的指標則其位址會被加 2
  - …. 依此類推 ….

2 x 16 LCD 模組顯示位址

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
第一行	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0AH	0BH	0Ch	0Dh	0Eh	0Fh
第二行	40h	41h	42h	43h	44h	45h	46h	47h	48h	49h	4AH	4BH	4Ch	4Dh	4Eh	4Fh

- DDRAM Address 控制指令為 80h，則游標位置控制程式可用 C 寫成

```
//*********************************************
//      Set Cursor position on LCD module
//       CurY = Line (0 or 1)
//       CurX = Position ( 0 to 15)
//
void LCD_Set_Cursor (unsigned char CurY, unsigned char CurX)
{
        WriteCmdLCD ( 0x80 + CurY * 0x40 + CurX) ;
        LCD_S_Delay( ) ;                                    // 40uS Delay
}
```

將ROM指標的字串送到LCD模組

```
//*************************************************
//            Put a ROM string to LCD Module
//*************************************************
void putrsLCD( const rom char *Str )
{
  while (1)
  {
      Str_Temp = *Str ;

          if ( Str_Temp != 0x00 )                  // 字串的結尾?
            {
                  WriteDataLCD ( Str_Temp ) ;      // 寫一個Byte到LCD模組
                  Str ++ ;                         // 指標指到下一個Byte
            }
          else
                  return ;                         // 跳出迴圈
  }
}
```

練習一

- 利用 WAP_LCD.C 所提供的 LCD 驅動函數撰寫
  - 將 “WAP-002 Workshop” 的字串從 ROM 讀出並顯示在 LCD 螢幕的第一行位置
  - 將你的英文名字顯示在第二行的起始位置
  - 將一數目字從 0 到 9 順序顯示在 LCD 第二行最後一個位置，間隔時間 0.5 Sec

```
WAP-002 Workshop
Richard Yang   7
```

LCD 顯示幕

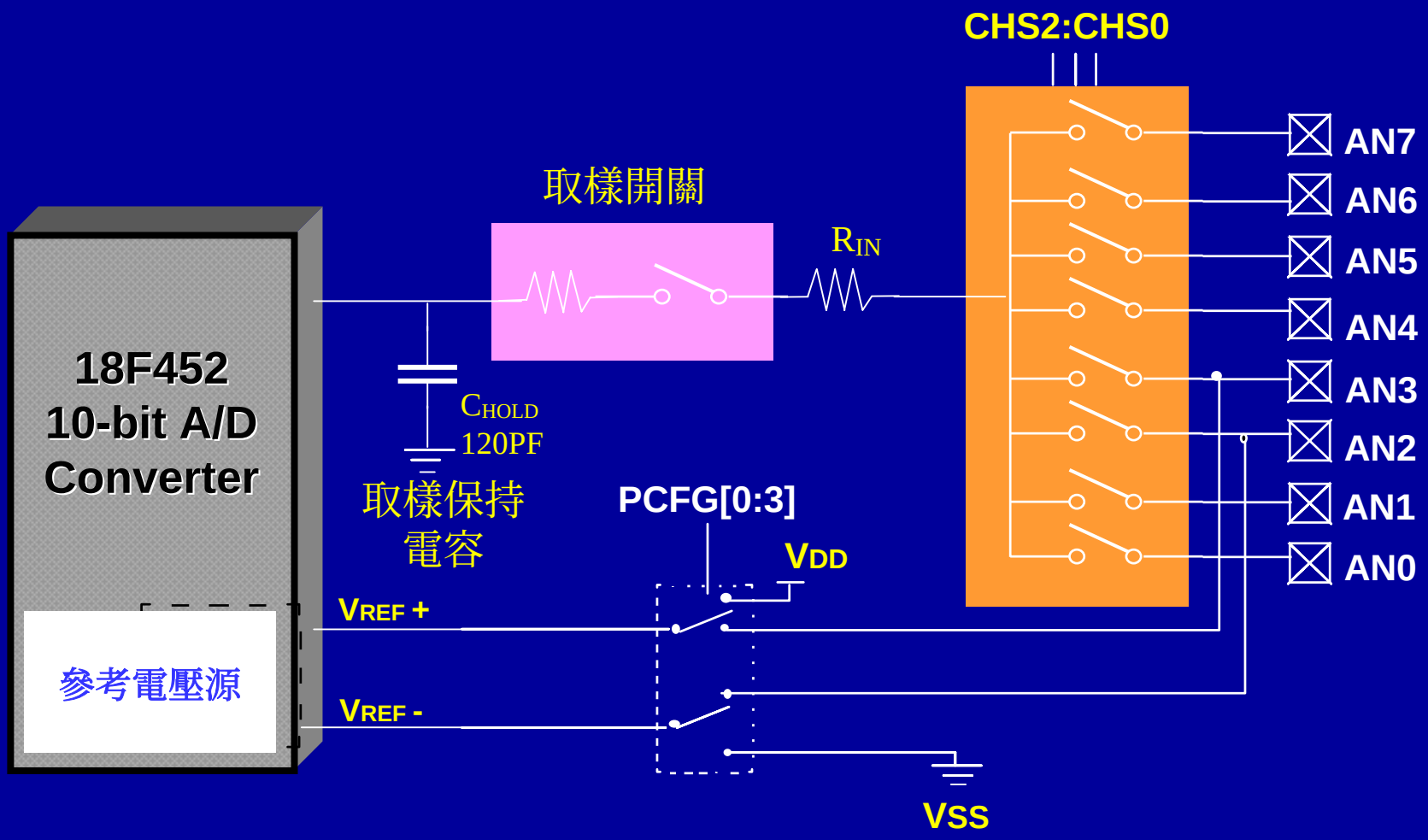
第三章
溫度的量測

1. 10-bit A/D 轉換器
2. 量測類比式溫度輸出
3. 結構變數
4. 量測數位式溫度輸出
5. Timer 1

10-bit A/D 轉換器

- 8組類比轉換多工輸入選擇，10 bits 解析度
- 類比輸入取樣時間：20 µS (輸入阻抗<10K)
- 類比輸入轉換時間：19.2 µS (12 T_{AD})
- 10-bit 解析度時，只有一位元的誤差
- 允許使用外部參考電壓： VREF+ & VREF-
- 轉換的結果允許自動向左、向右對齊修正
- 完整的轉換時間共須 39.2 µs
  - 如輸入腳位固定，其轉換時間只需：29.2 µs

10-bit A/D 方塊圖



A/D 控制暫存器

See Data Book

ADCON0 Register

ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	---	ADON
bit7							bit0

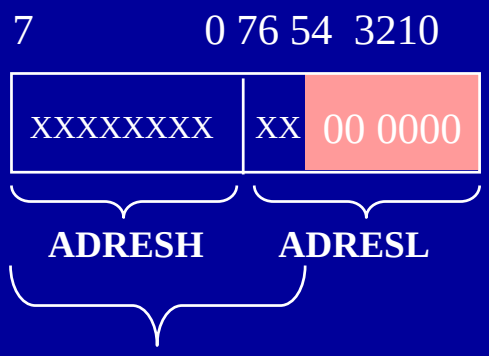
ADCON1 Register

ADFM	ADCS2	-----	-----	PCFG3	PCFG2	PCFG1	PCFG1

10-bit Result

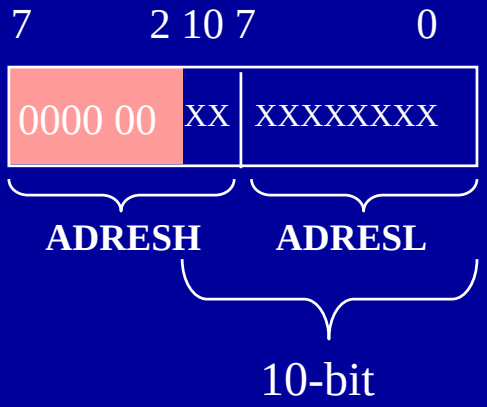
A/D轉換結果
向左、向右調整

ADMF=0



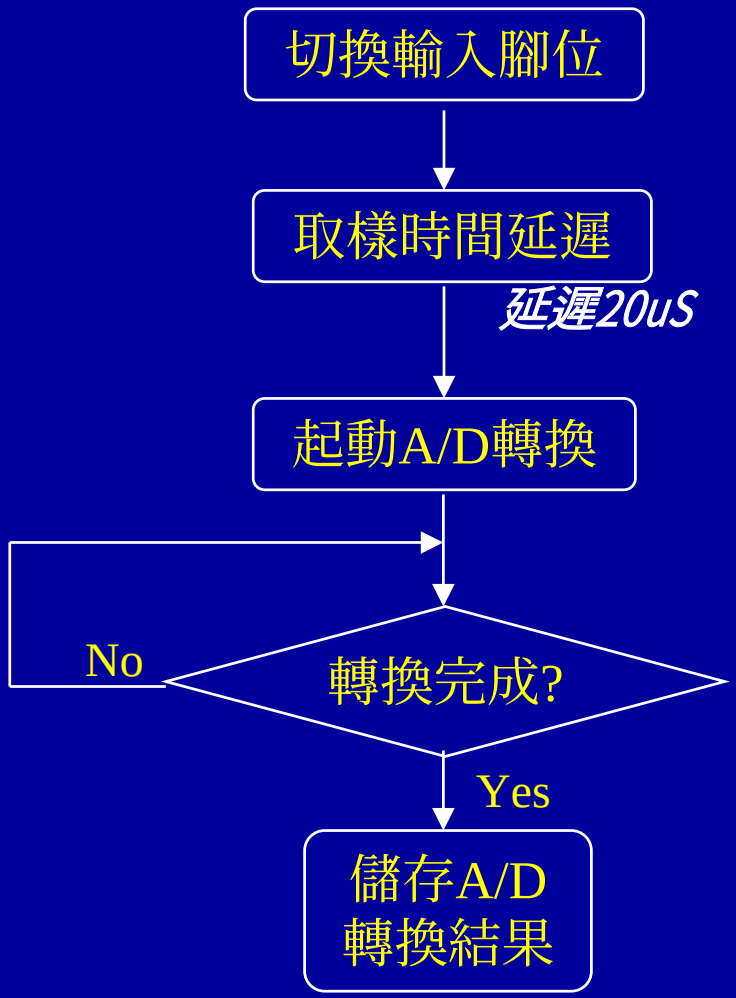
向左靠齊

ADMF=1



向右靠齊

A/D 轉換基本流程



量測類比式溫度輸出

TC1047A 溫度轉類比電壓輸出

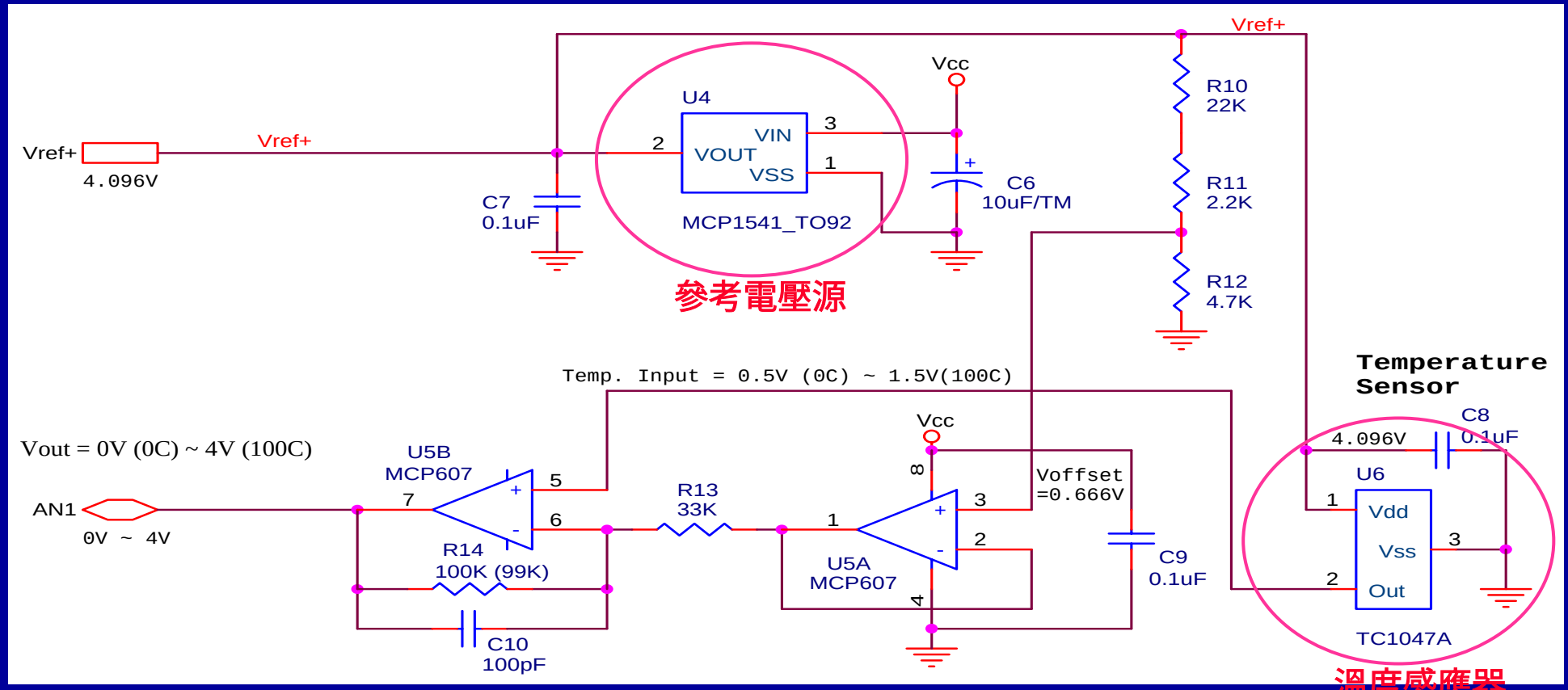
TC1047A – 溫度對電壓轉換器

- 溫度轉換成線性類比電壓輸出：10mV/ºC
- 量測溫度範圍
  - -40ºC ~ +125ºC (精確度 +-1ºC$_{(Typ)}$)
  - 25ºC ~ +125ºC (精確度 +-0.5ºC$_{(Typ)}$)
  - 0ºC 輸出電壓為 500mV，100ºC 時輸出為1.5V
  - 輸出電壓為：(10mV x 目前溫度) + 500mV
- 工作電壓：2.5V ~ 5.5V
- 消耗電流：35uA
- SOT-23 3-pin 包裝

TC1047A 輸出電壓

- 欲量測的溫度從：0ºC~100ºC
  - 直接的電壓輸出為 500mV~1500mV
  - 以 PIC16F452 的 A/D 直接轉換只能得到部份的值，且解析度會不足 (以 Vdd 為參考電壓)
  - 直接轉換後的輸出值為：0x66(0ºC) ~ 0x133(100ºC)
- 需設計一放大與位準轉換電路以符合18F452的A/D輸入的需求
  - 將 TC1047A 的輸出從 500mV ~ 1500mV → 0V ~ 4.0V
  - 因輸出有低到 0V，所以使用軌對軌輸出的運算放大器
  - 18F452 採外部 4.096V 參考電壓輸入
  - 轉換後的輸出值為：0x00(0ºC) ~ 0x3E8(100ºC)

TC1047A – 應用電路範例



Vout= [-Voffset (R14)/R13] + [Temp. Input (R14+R13)/R13]

Temperature = 0C ,Vout (0C)= -0.666V * 3 + 0.5V * 4 = 0V

Temperature = 100C, Vout = -0.666V * 3 + 1.5V * 4 = 4V

讀取TC1047A的溫度值

Read_Tmp.C

```
int Read_TC1047_Temperature (void)
{
        int  AD_Temp;

        if (Flagbits.SW2_Flag) ADCON0bits.CHS0=0;        // Read A/D from CH0 (VR)
        else ADCON0bits.CHS0=1;                          // Read A/D from CH1 (T2)

        for (AD_Temp=0;AD_Temp<5;AD_Temp++);             // Delay 20uS for CH change

        ADCON0bits.GO=1;                                 // Start to convert the A/D
        while (ADCON0bits.GO);                           // Waiting A/D until done
        AD_Temp = ReadADC( );                            // Get 10 bits A/D result
        return AD_Temp;
}
```

加入讀取VR的電壓值作為大溫度範圍的測試

TC1047A 溫度的顯示

- 輸出電壓轉換為 0V ~ 4.0V
- 使用 MCP1541 為 A/D 的參考電壓源, 則在 10 bit A/D 的轉換結果剛好為 0 ~ 999 (0x00~0x3E8)
- 每 ºC 的移動值為 10 個 LSB , 只要將 A/D 的結果除以 10 便得到溫度的整數值
- 若使用 itoa( ) 將 A/D 的結果轉成 ASCII code 後， 可再最小位數前加一個小數點； 如此可顯示小數點後一位的精確度

基本資料結構型態

結構變數

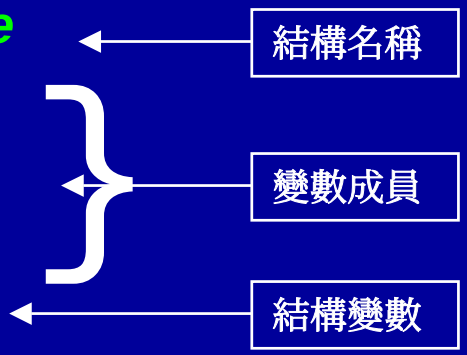
位元結構

共用型態

結構型態 (Structures)

- 結構是可以把不同型態的資料收集在一起當作一個整體, 可以用"結構變數名稱.成員"的名字來指定結構中的某一成員
- 結構變數的位址可以用 & 運算子取得
- 使用結構的場合
  - 成員之中具有各種不同的資料型態 (型態相同可用陣列)
  - 多樣化變數宣告
  - 單一 bit, 或數 bit 的資料運算

```
struct struct-name
{
    type member1;
    type member2;
    . . .
} variable-name;
```



使用結構變數

- 欲使用結構內的成員，可使用“.”來組成:

  variable-name.memberx （結構變數．成員）

```
struct Comm_protocol
{
   char ID[6];
   char Data[10];
   char  Message[20];
   unsigned int CRC;
   unsigned char Repeat;
} Rec_Fram;
        :
        :
unsigned char j;
for(j=0;j<20;j++)
{
writeUSART(Rec_Fram.Message[j]);
}
```

Comm_protocol 在 RAM 的排列

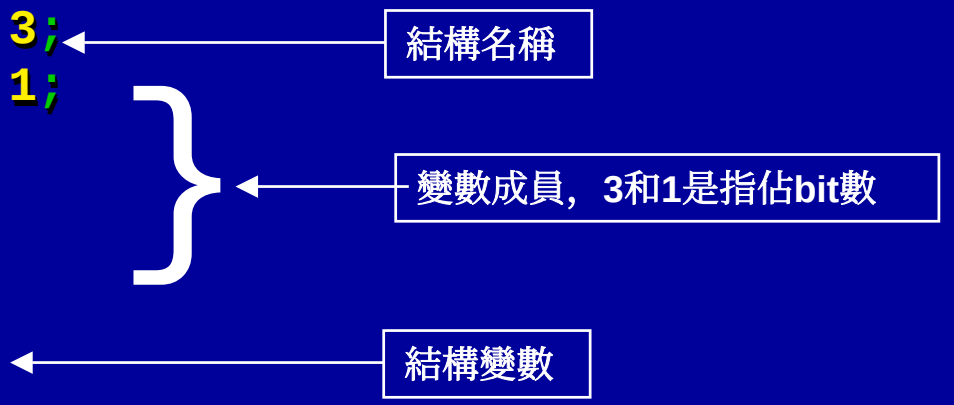
Comm_protocol 在 RAM 的排列

Rec_Fram	
成員名稱	資料長度
ID	6 Bytes
Data	10 Bytes
Message	20 Bytes
CRC	2 Bytes
Repeat	1 Bytes

使用位元結構（bit）

- 位元結構是集合獨立位元的一種特殊結構
- 該結構中的組成成員，最大值為一個位元組(byte)
- 結構中的組成成員可以是一個個單獨的位元(bit)或數個位元(group of bits)
  - 位元成員的存取、標記和一般的結構變數相同

```
struct struct-name
{
    unsigned member1 : 3;
    unsigned member2 : 1;
    . . .
} variable-name;
```



定義位元結構（範例）

```
extern volatile near unsigned char PORTB;   // 宣告PORTB是一個Byte
extern volatile near union  {                // 宣告PORTBbits為一位元結構的共用
    struct {                                 // 形態(union),位址在Access RAM
        unsigned RB0:1;              // 定義PORTB的標準功能
        unsigned RB1:1;
        unsigned RB2:1;
        unsigned RB3:1;
        unsigned RB4:1;
        unsigned RB5:1;
        unsigned RB6:1;
        unsigned RB7:1;
    } ;
    struct {
        unsigned INT0:1;             // 定義PORTB的另外功能
        unsigned INT1:1;
        unsigned INT2:1;
        unsigned CCP2:1;
    } ;
} PORTBbits ;
```

摘錄自“P18C452.H”檔

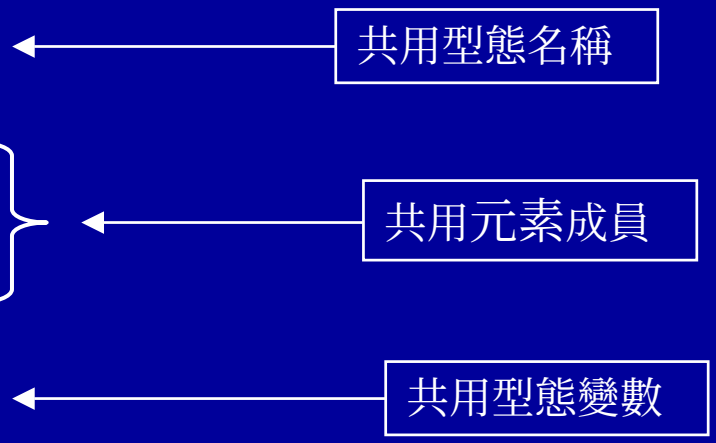
使用位元結構（範例）

```
PORTB=0x34;                              /* 記著！ PORTB & PORTBbits
                                         的位址定義是在 p18C452.asm中*/
        :
PORTBbits.RB7=1;                         // RB7 輸出Hi
PORTBbits.RB6=!PORTBbits.RB6;   // RB6 輸出轉態
        :
        :
if (PORTBbits.INT0)                      //  測試INT0腳電壓?
    {
        PORTAbits.RA0=1;
        Nop();
        PORTA >>= 1;
    }
else PORTA=0;
```

共用型態（union）

- 共用型態(union)，可使幾種不同的資料型態的變數共用一塊記憶空間
  - 共用型態(union)使用方式類似結構型態(Structure)
  - 共用型態(union)內的變數稱為共用元素
  - 共用型態常使用於資料轉換
- 編譯器會根據共用元素中佔記憶空間的最大者來分配記憶空間
  - 可同時宣告各種不同型態的變數

```
union union-name
{
    type member1;
    type member2;
    . . .
} variable-name;
```



共用型態的資料架構

```
union EE_tag
{
    int  Word;
    char Bytes[2];
} TC_74;
```

Result	
Byte[0]	Byte[1]
RAM0	RAM1

EE_tag 佔 2 個 Bytes

例：

```
union
{
        int Word;
        char Bytes[2];
} EE_Read_Data;

for (i=0;i<5;i++)
    {
        EE_Addr=i;
        TC_74.Word = EERando
        if ( EE_Read_Data.Result >= 0 )
        Secu_Code[i]= TC_74.Bytes[0];                    }
```

EERandom Read( )讀進來的資料 為“int” 型態，利用“union”拆成兩個“char”後，只擷取其中的低位元組“Low-byte”

** EERandomRead( ) 的傳回值要 >= 0 才表示讀取中無錯誤 !!

合併使用 結構型態 & 共用型態

- 一個結構型態或共用型態的宣告內，也可含有其它的共用型態或結構型態

```
union FPvar
{
   float FPNum;                    //floating point access
   struct
   {
      unsigned char Arg0;          //argument byte 0 access
      unsigned char Arg1;          //argument byte 1 access
      unsigned char Arg2;          //argument byte 2 access
      unsigned char Exp;           //exponent byte access
   }
} Foo;

Foo.FPNum = 3.14159;
Exponent = Foo.Exp - 0x7F;
```

Byte 與 bit

```
near union
{
        unsigned char Count;

        struct {
                unsigned B0:1;
                unsigned B1:1;
                unsigned B2:1;
                unsigned B3:1;
                unsigned B4:1;
                unsigned B5:1;
                };

} Bz=0;
```

Bz.Count是指到Byte
Bz.B1是指到bit

Main.c 的中斷程式部份

```
{
        Buzzer_Count--;
        Bz.Count++;
        if (Flagbits.Buzzer_Fast_Flag==1)               // Check the alarm with fast mode
        {
                if (Bz.B1==1) TRISCbits.TRISC2=0;       // Turn On the PWM1 output (Buzzer)
                else TRISCbits.TRISC2=1;                // Turn off the PWM1 output (Buzzer)
        }
        else if (Flagbits.Buzzer_Mid_Flag==1)           // Check the alarm with slow mode
        {
                if (Bz.B2==1) TRISCbits.TRISC2=0;       // Turn On the PWM1 output (Buzzer)
                else TRISCbits.TRISC2=1;                // Turn off the PWM1 output (Buzzer)

        }
        else if (Flagbits.Buzzer_Slow_Flag==1)          // Check the alarm with slow mode
        {
                if (Bz.B4==1) TRISCbits.TRISC2=0;       // Turn On the PWM1 output (Buzzer)
                else TRISCbits.TRISC2=1;                // Turn off the PWM1 output (Buzzer)

        }

}
```

量測數位式溫度輸出

TC74 SMBus / I^2C 介面

I^2C 介面存取

TC74 的基本規格

- SMBus / I²C 介面，傳送速率100KHZ(Max.)
- 量測溫度範圍
  - +25ºC ~ +85ºC (精確度 +-2ºC)
  - 0ºC ~ +125ºC (精確度 +-3ºC)
- 內建溫度二極體，Delta-Sigma A/D 轉換器
- 轉換速率：每秒 8 次
- 工作電壓：2.7V ~ 5.5V
- 消耗電流：200uA，靜態電流 5uA
- SOT-23 5-pin 包裝

TC74 的應用

- CPU、硬碟溫度保護
- 電源供應器溫度保護
- PC週邊介面卡、筆記型電腦溫度保護
- 自動溫控系統
- 基板溫度量測、系統保護
- 一般室溫量測

讀取 TC74 Configuration Data

Command	Code	Function
RTR	00h	Read Temperature (TEMP)
RWCR	01h	Read/Write Configuration (CONFIG)

動作順序

先送出 0x01 讀取狀態

再送出 0x00 讀取溫度

Bit	POR	Function	Type	Operation
D[7]	0	STANDBY Switch	Read/ Write	1 = standby, 0 = normal
D[6]	0	Data Ready *	Read Only	1 = ready 0 = not ready
D[5]- D[0]	0	Reserved - Always returns zero when read	N/A	N/A

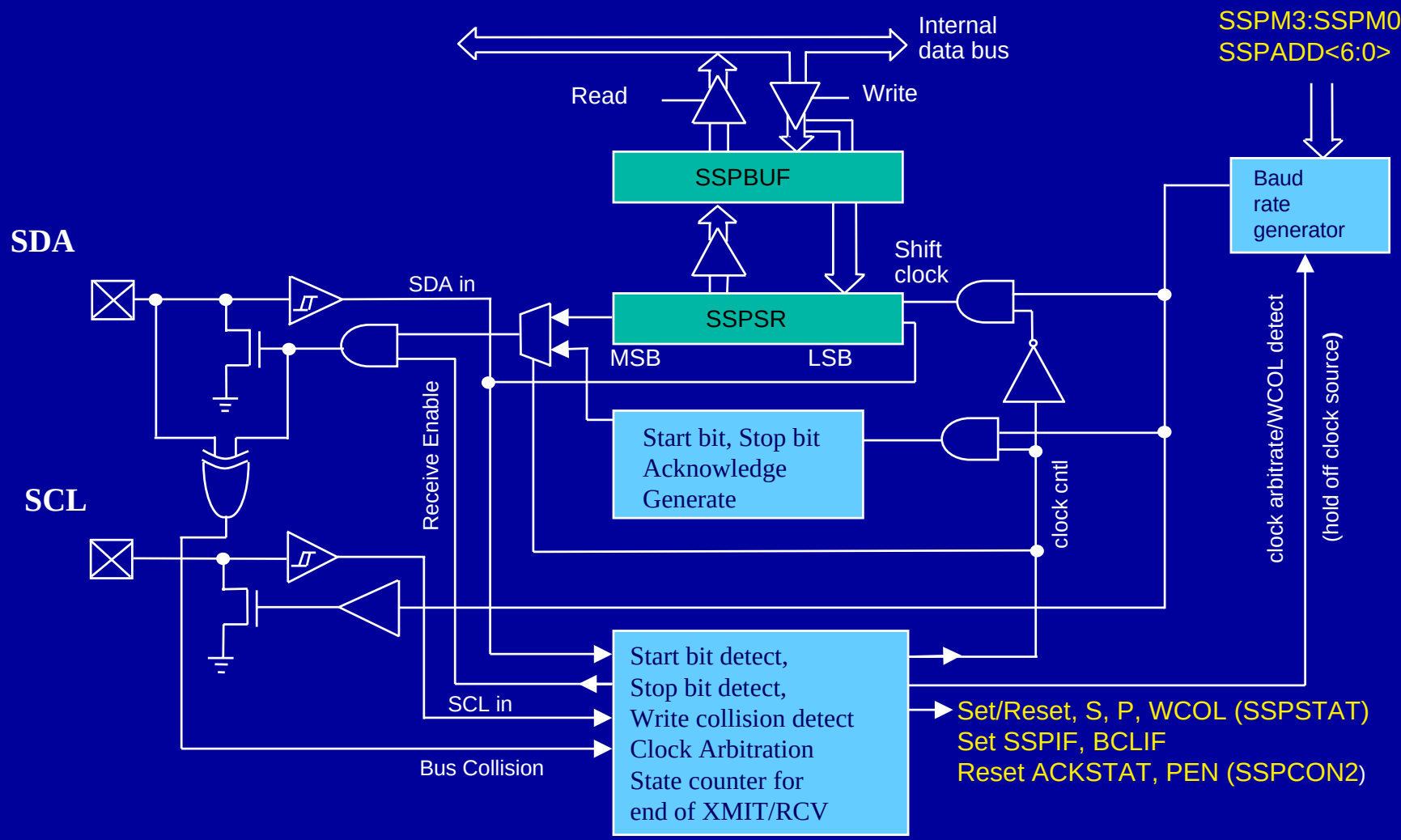
讀取狀態資料後檢查 Config. 的 **Bit 6** 以了解溫度轉換是否完成

讀取溫度資料

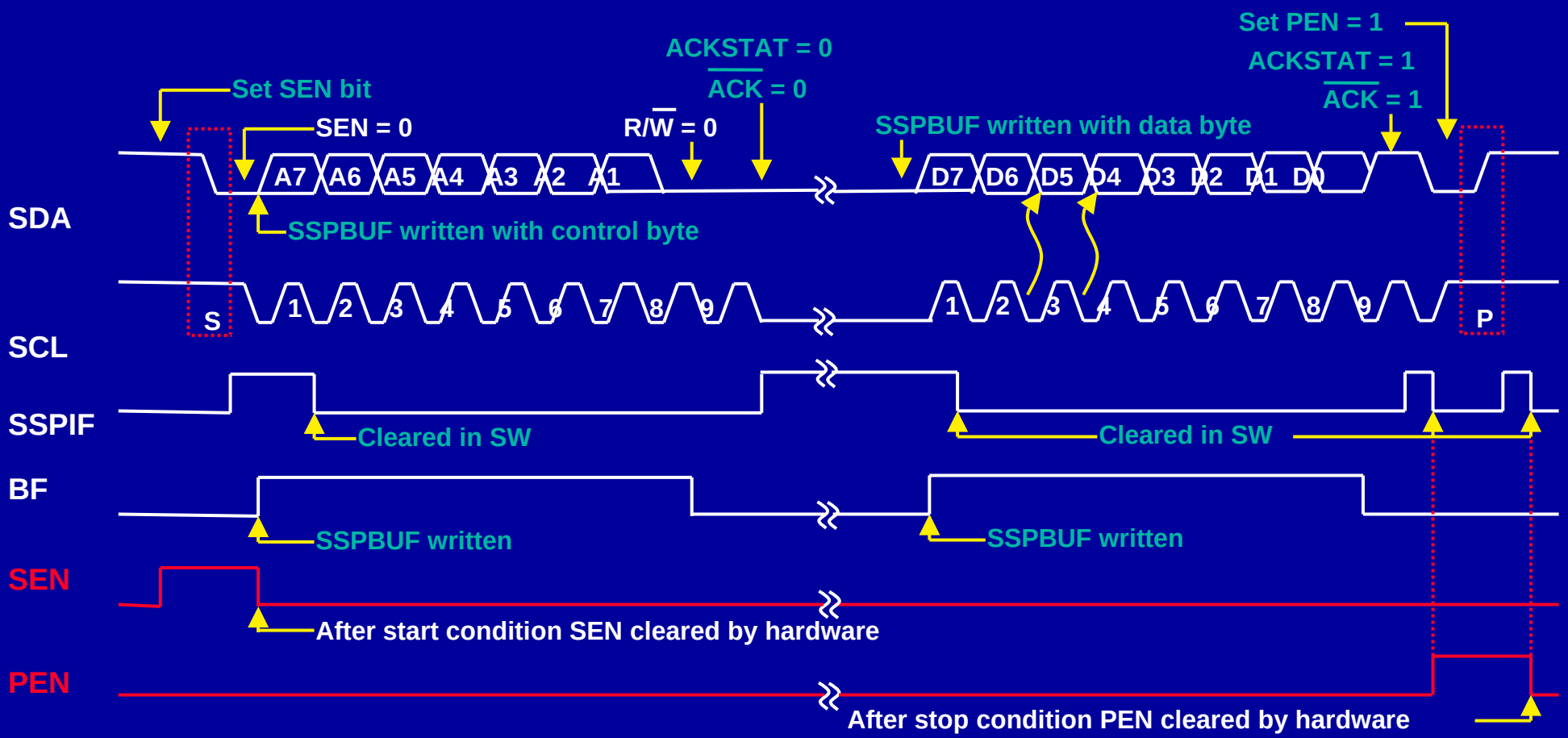
- 溫度採 8-bit 輸出，負溫採 2‘S 格式
- 1 LSB 代表 1ºC 的溫度變化

Actual Temperature	Registered Temperature	Binary Hex
+130.00°C	+127°C	0111 1111
+127.00°C	+127°C	0111 1111
+126.50°C	+126°C	0111 1110
+25.25°C	+25°C	0001 1001
+0.50°C	0°C	0000 0000
+0.25°C	0°C	0000 0000
0.00°C	0°C	0000 0000
-0.25°C	-1°C	1111 1111
-0.50°C	-1°C	1111 1111
-0.75°C	-1°C	1111 1111
-1.00°C	-1°C	1111 1111
-25.00°C	-25°C	1110 0111

I²C Master 模式方塊圖



I²C Master 模式下
發送資料的時序

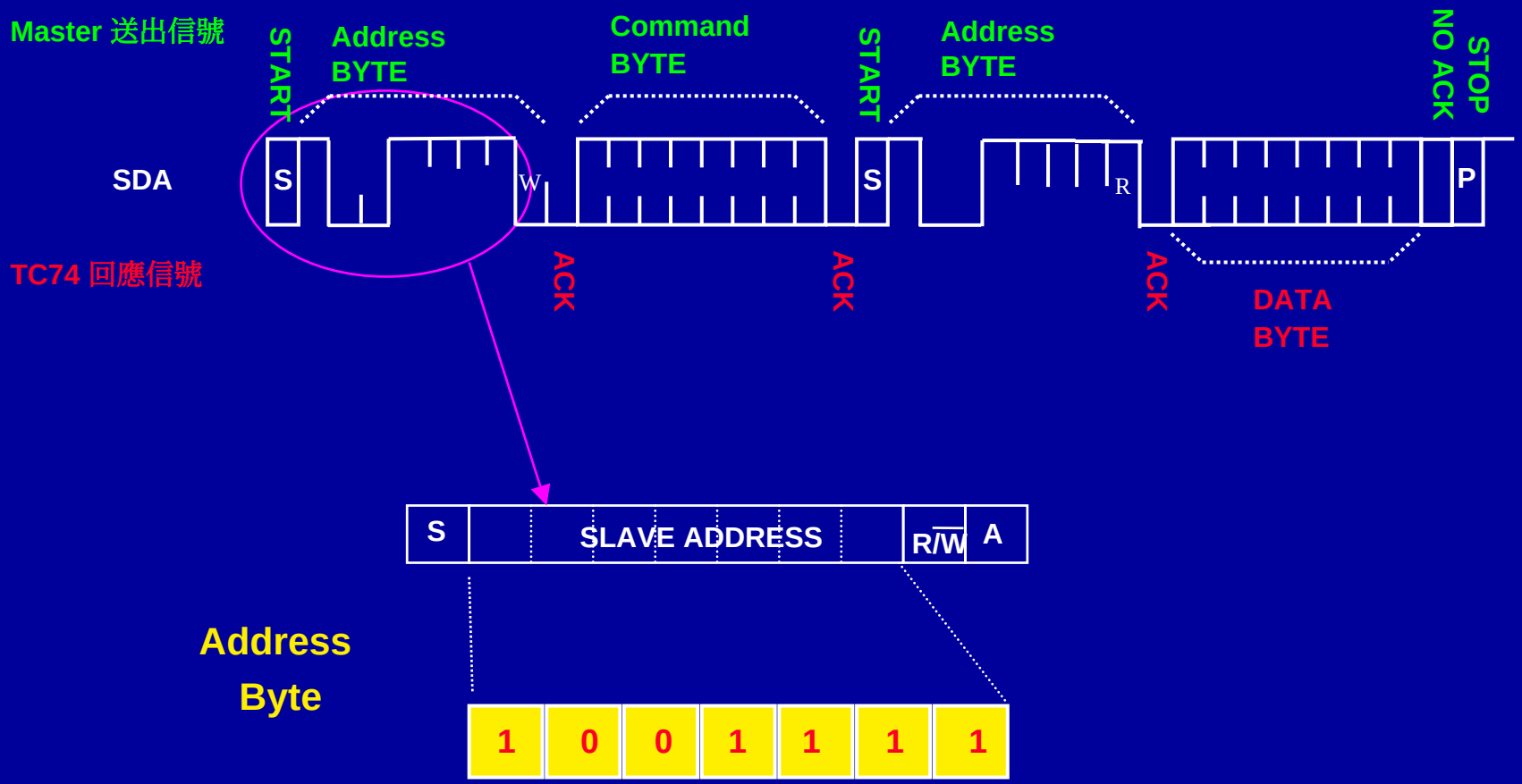


TC74-Ax

- TC74-Ax 的讀取方式
  - TC74 為使用 SMBus / I^2C 通信協定的溫度偵測器
  - TC74 的基本位址為 1001xxx0
    - xxx 為位址位元，範圍由 0 .. 7，料號中的 Ax 表示其使用到的位址 (TC74-A0 .. TC74-A7)
    - TC74-A7 表示使用的是 10011110 為 I^2C 位址
  - TC74 的讀 / 寫 格式與一般標準的 EEPROM 相同，只是位址不同而已

TC74-A7 的 I²C時序

讀取資料方式與 24LCxx 的 EEPROM 類似



I²C 的相關函數

- MPLAB C18 Library 提供的基本 I²C Functions

Function	Description
AckI2C	Generate I²C bus *Acknowledge* condition.
CloseI2C	Disable the SSP module.
DataRdyI2C	Is the data available in the I²C buffer?
getcI2C	Read a single byte from the I²C bus.
getsI2C	Read a string from the I²C bus operating in master I²C mode.
IdleI2C	Loop until I²C bus is idle.
NotAckI2C	Generate I²C bus *Not Acknowledge* condition.
OpenI2C	Configure the SSP module.
putcI2C	Write a single byte to the I²C bus.
putsI2C	Write a string to the I²C bus operating in either Master or Slave mode.
ReadI2C	Read a single byte from the I²C bus.
RestartI2C	Generate an I²C bus *Restart* condition.
StartI2C	Generate an I²C bus *START* condition.
StopI2C	Generate an I²C bus *STOP* condition.
WriteI2C	Write a single byte to the I²C bus.

I²C EEPROM 的相關函數

- MPLAB C18 Library 提供的整合式 I²C Functions
  - 這些 Functions 提供批次處理的功能, 只要提供正確的位址資訊及欲寫入的資料即可
  - EERandonRead 及 EECurrentAddRead 會將讀入的資料及執行結果放在一個整數 (int) 的資料型別中傳回

Function	Description
EEAckPolling	Generate the Acknowledge polling sequence.
EEByteWrite	Write a single byte.
EECurrentAddRead	Read a single byte from the next location.
EEPageWrite	Write a string of data.
EERandomRead	Read a single byte from an arbitrary address.
EESequentialRead	Read a string of data.

EERandonRead 函數

- EERandonRead 的使用範例與傳回結果

```
unsigned int temp ;                    // 宣告一個整數值來接收傳回值
temp = EERandonRead(0xA0,0x30)         // 讀取位址 0x30 的資料
```

- 如果讀取的過程無誤, 則傳回值的 High Byte 會是 0 而 Low Byte 則包含讀入的資料 , 此時的傳回值為正數
- 若讀入的過程中有錯誤, 則傳回值的狀態值會放在 High Byte , 此時可檢查整個傳回值的內容來得知錯誤訊息 , 此時的傳回值為負值
  - -1 Bus Collision Error Happened
  - -2 No ACK Error Happened
  - -3 Write Collision Error Happened

定義 TC74-A7 的常、變數

Read_Tmp.C

```
union
{
        int             Word ;
        unsigned char   Bytes[2] ;
        struct  {
                unsigned : 6 ;
                unsigned  BitD6 : 1 ;
                };
} TC_74 ;

#define  TC74_Addr      0b10011110      // Define the TC74-A7 address
#define  TC74_RWCR      0x01            // Define the Read/Write Configuration
#define  TC74_RTR       0x00            // Define the read temperature command
```

TC_74.Bytes[0] → 為 8-bit 的溫度資料
TC_74.BitD6 → 為判斷溫度是否轉會完畢

讀取 TC74-A7 的溫度資料

Read_Tmp.C

```
unsigned Read_TC74_Temperature(void)
{
    TC_74.Word = 0 ;
    TC_74.Word = EERandomRead(TC74_Addr,TC74_RWCR);    // Read Status from TC74
    if ( TC_74.BitD6 )                                 // b6 =1 , Read
    temperature
    {
        TC_74.Word=EERandomRead(TC74_Addr,TC74_RTR);
        if ( TC_74.Word >= 0 ) return TC_74.Word;
        else return -1;                                // Read Fail, return (-1)
    }
    else return -2;                                    // b6=0, return (-2)
}
```

溫度顯示在 LCD

- 溫度值為一16進制值需經數值轉換後才能在 LCD 顯示
  - 數值必須轉換為 ASCII code 的字元或字串後才能顯示於終端機或 LCD
    - MPLAB C18 提供此轉換函數 (itoa( ))
    - Itoa (i , *Str) 將整數 i 轉換成十進制型態的 ASCII code 字串後置於指標變數 Str
    - Str 通常為一個在 Data Memory 的陣列起始位址

變數值為 $1000_{(16)}$ → 用十進制表示時為 $4096_{(10)}$, itoa( ) 會將轉換結果以十進位 ASCII code 的格式輸出“4096”

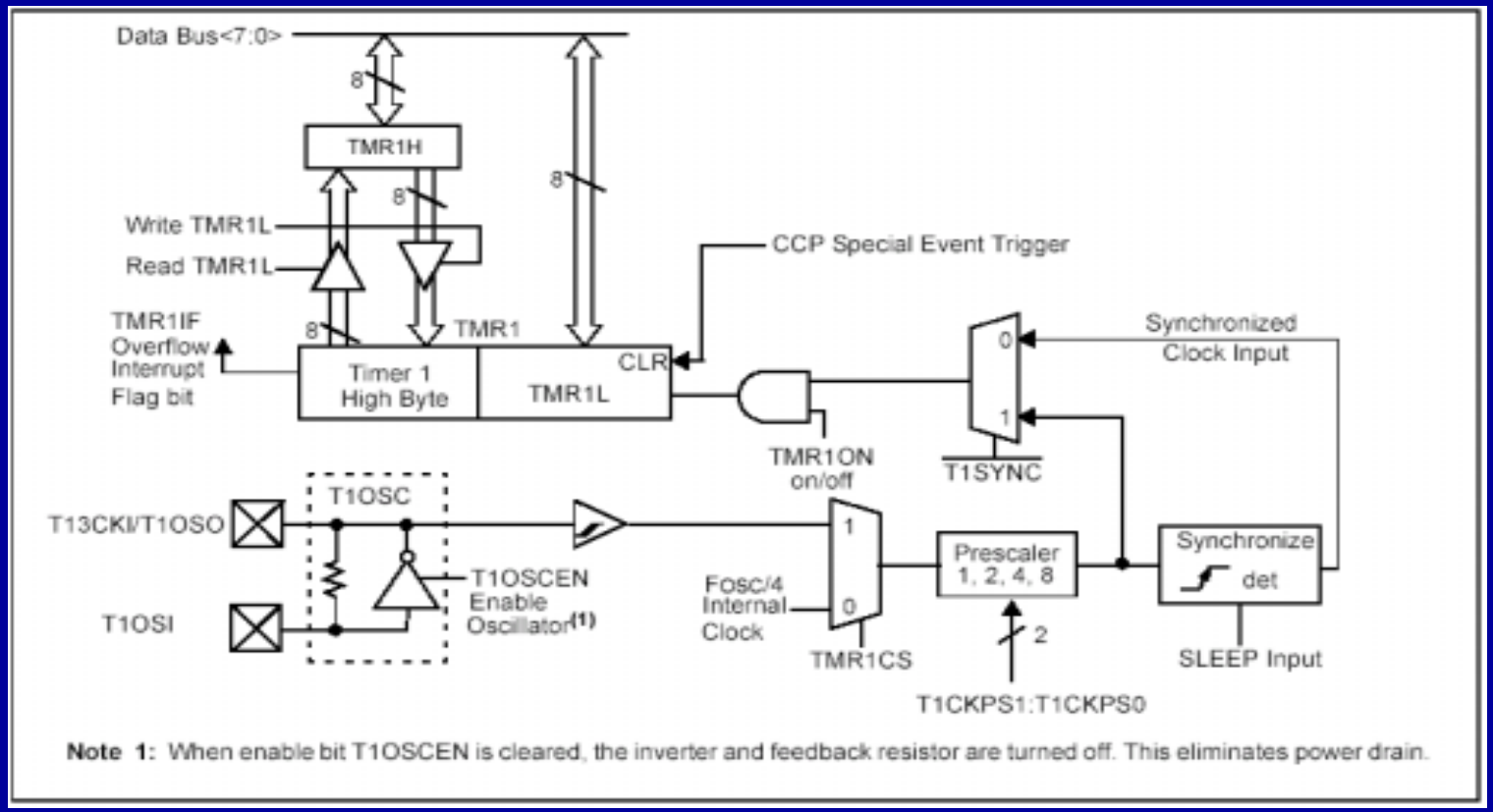
0x34 , 0x30 , 0x39 , 0x36 , 0x00

將數值轉成 ASCII 字串的函數

- btoa( )
  - 將8位元有號整數轉換成10進制的 ASCII 數字字串後存到指標指到的位址
  - 例: 0x80 -->”-128”， 100-->”100”， 199-->”-57”
- itoa( )
  - 轉換16位元的有號整數成10進制的 ASCII 數字字串
  - 例: 0x1000-->”4096”， 1000-->”1000”
- ltoa( )
  - 轉換32位元的有號整數成10進制的 ASCII 數字字串

Timer 1 方塊圖

- Timer1 為 16 bit 的計時器, 計時器的結果存於 TMR1H 及 TMR1L



Timer1 的動作及相關函數

- Timer1 的動作說明
  - Timer1 的讀寫模式可規劃為 8 bit 或 16 bit 存取模式
    - 8 bit 存取模式時 TMR1H 和 TMR1L 是獨立被讀寫的
    - 16 bit 存取模式時 TMR1H 會於 TMR1L 被讀寫時同時將 Timer1 的 High Byte 讀入 TMR1H 或由 TMR1H 載入 Timer1 的 High Byte
      - Write to Timer1： 先 TMR1H 再 TMR1L
      - Read from Timer1 : 先 TMR1L 再 TMR1H
  - 相關的函數 (含入檔 timers.h)
    - Void OpenTimer1 (unsigned char config)
    - Unsigned int ReadTimer1 (void)
    - Void WriteTimer1 (unsigned int timer)

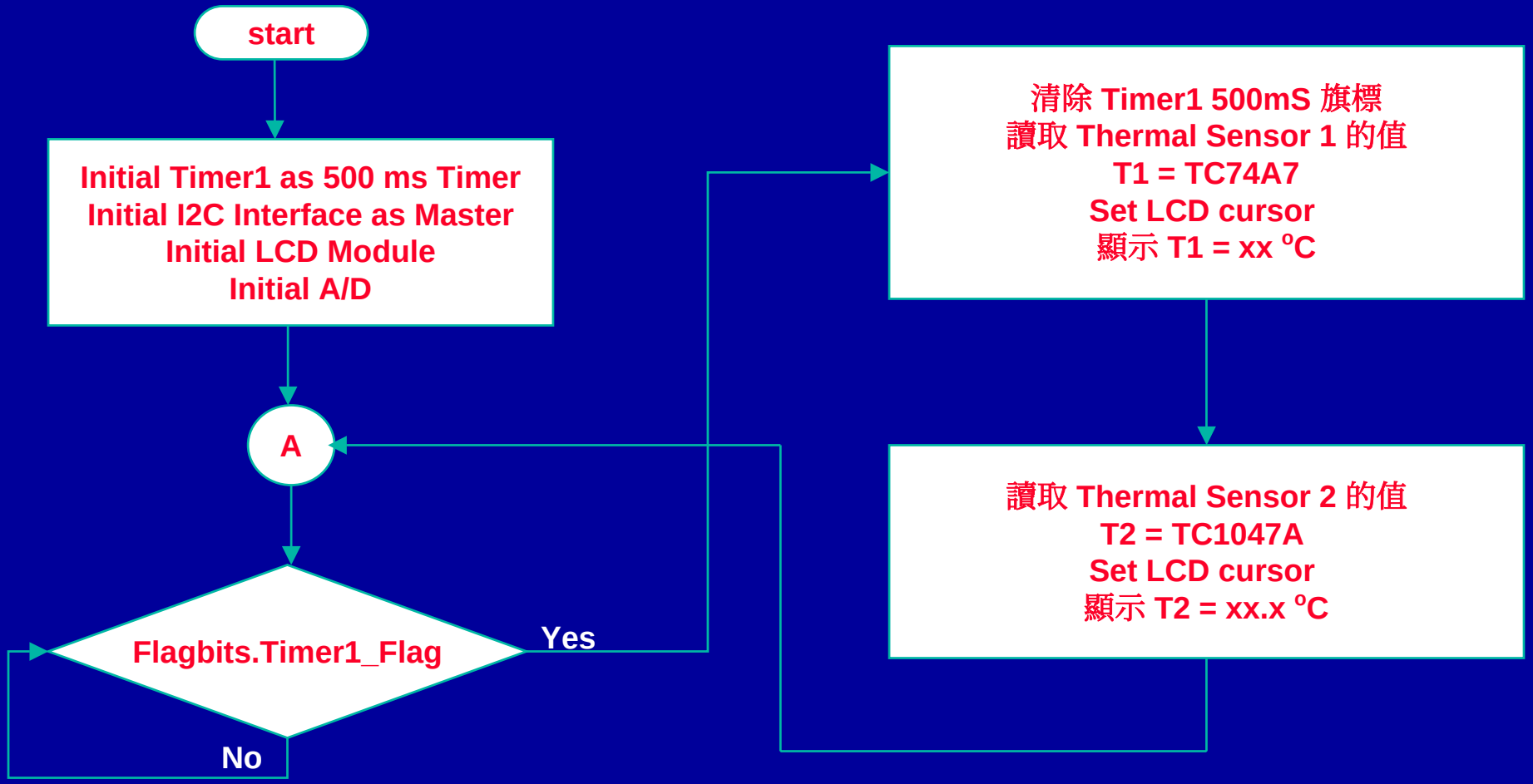
Timer1 的溢位發生

- 當 Timer1 的計時值累積至 0xFFFF 後將產生溢位而變成 0x0000 後繼續上數
  - 此時 TMR1IF 位元 將被設定成 “1”
  - TMR1IF 位元的位置在 PIR1 暫存器中
  - 可以用 if (PIR1bits.TMR1IF) 來判斷 Timer1 的溢位是否已經發生 (Polling 中斷的做法)

```
If ( PIR1bits.TMR1IF )
    {       //      Do your job here        }
```

  - 若 TMR1IE 位元也被設定為 1 , 則會有中斷產生

Exer1 的流程圖



採用 Polling 0.5 Sec旗號的做法

練習二的程式檔案

- Exer1 將由下列原始程式組成
  - Main.h
  - Main.c
  - F18_Conf.c (只做 Configuration Word 設定)
  - Init_MCU.c
  - Read_Tmp.c
  - WAP_LCD.c
- 每個 .c 的程式都使用 Main .h 檔來對所有自訂的函數作原型宣告的工作

練習二的程式
Main.h
WAP_LCD.c

- Main.h
  - 程式中必須的含入檔 (include files)
  - 自訂函數的原型宣告 (Prototype)
- WAP_LCD.c
  - LCD 的顯示函數

練習二的程式 Init_MCU.c

- 請使用 OpenTimer1 ( ) , WriteTimer1( ) 來將 Timer1 規劃為一個 500 ms 中斷一次的計時器
  - 使用 Timer1外部的石英振盪器 (32768Hz 的 XTAL)
  - 將中斷打開 – RCON 暫存器的 IPEN 位元要設為 1，如此才有高/低優先權的區分
  - IPR1 中的 TMR1IP 設為 “0”， Timer1 的中斷為低優先權
  - 記得！將 Timer1 設為 16 bit RW 模式
  - 因為 Timer1 使用的是 32768Hz 的振盪器, 故寫入值若為 (65536 - 16384) 則可讓 Timer1 每 500ms 後產生中斷

練習二的程式 Read_Tmp.c

- 讀取 TC74-A7 溫度值
  - 用內建 EEPROM 的函數讀取溫度值
  - 使用函數 “EERandomRead( )
- 讀取 TC1047A 溫度值
  - 用18F452內建的 10-bit A/D 轉換器讀取
  - 參考電壓使用外部 4.096V

練習二的程式 Main.c (Timer1 中斷部分)

- 低優先權的 ISR 名稱為 isr_low , 請在 Timer1 中斷後進行下列的工作
  - 使用 WriteTimer1( ) 將 Timer1 的 500 ms 值重新載入設定
  - 清除 PIR1bits. TMR1IF , 讓 Timer1 可再次中斷
  - 設定 Flagbits. Timer1_Flag 已通知主程式
    - Flagbits 已被宣告為 " 位元結構 "

練習二的程式 Main.c （ 主程式部分 ）

- 每次 Timer1 中斷發生一次後就呼叫一次函數 LCD_Temp_Update( )
  - 請在 LCD_Temp_Update( ) 中 ….
    - 讀取 TC74-A7 的溫度值， 並顯示在 LCD
    - 讀取 TC1047A 的溫度值， 經轉換成字串型態並加入小數點後， 顯示在 LCD
  - 讀取 TC74 的函數使用 EERandomRead ( )， 可以偵測如 IC 不存在或異常的多種錯誤狀況

練習二

- 讀取兩個不同型態的溫度感應器並將其轉換後的溫度顯示在 LCD 顯示幕上
  - T1 為數位式溫度，範圍為兩位數
  - T2 為類比式溫度，範圍為 xx.x
  - 每隔 500mS 更新顯示的溫度

```
WAP-002 Exer 2
T1=24⁰ T2=24.5⁰
```

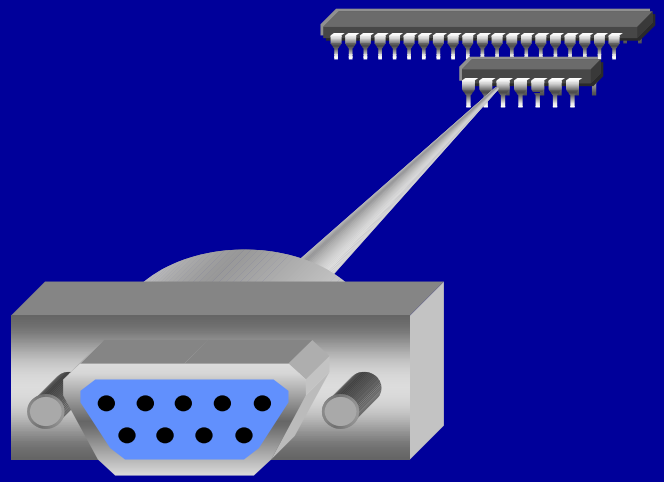
LCD 顯示幕

第四章
使用 VT-100 終端機

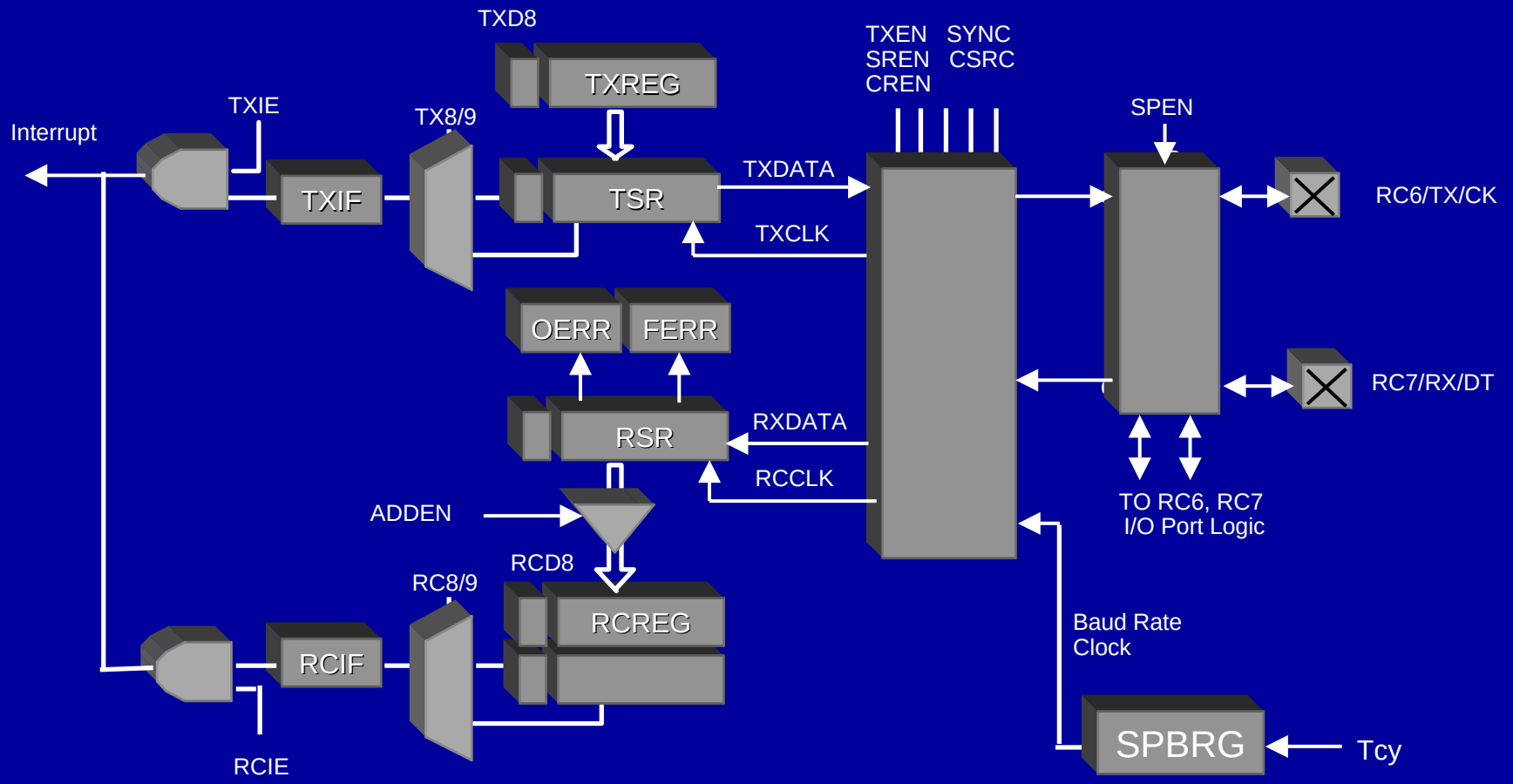
1. RS-232 串列通訊
2. VT-100 終端機
3. 指標型態的陣列
4. 溫度的顯示

18F452 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的中斷來源
- 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625 Kbps或高速模式2.5 Mbps
- 支援 9-bit 位址或資料的判斷模式

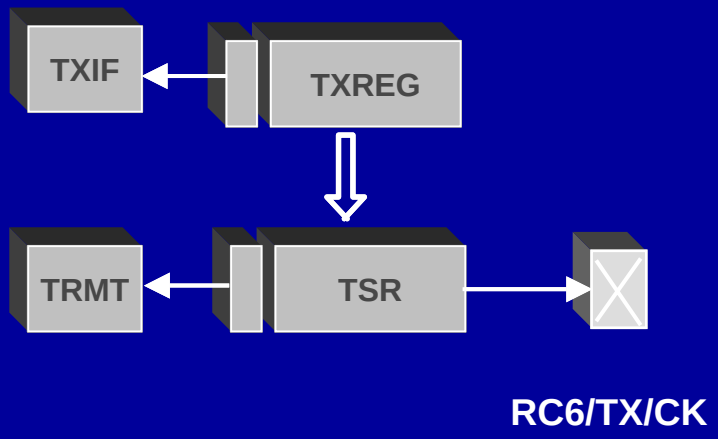


18F452 串列通訊方塊圖



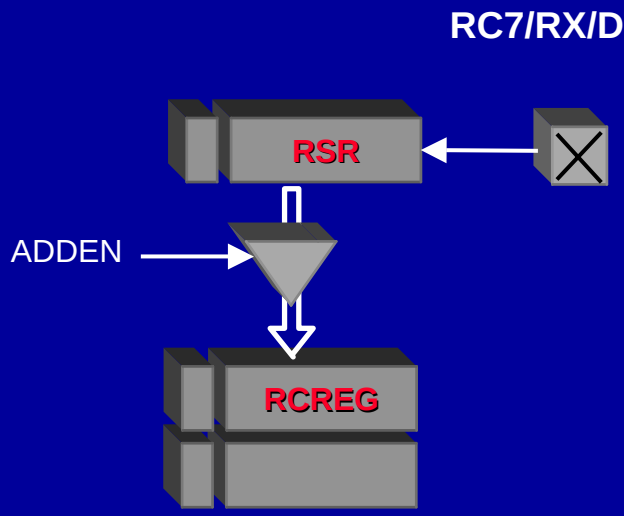
資料傳送 TXIF & TRMT 的動作

- 資料寫入 TXREG，TXIF = 0。TXREG 的資料被載到 TSR，TXREG 會空出，則 TXIF = 1
- TSR 的資料串列傳送完畢後，TMRT = 1；此時才可關閉 USART
- TXIF是可單獨使用，即使 USART 的 TX 中斷是關閉的 (TXIE=0)
- TXIF & TRMT 是完全反映硬體狀態的旗標 (Read Only) , 而檢查 TRMT 則可以用來確定所有資料完全被送出！ (RS485 or H/W Hand-Shake 有用)



接收中斷 RCIF 的動作

- RSR 為一移位器，接收8-bits串列資料
- 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



2 deep FIFO

常用的 USART 函數庫

- USART的原始程式放在:
  - c:\mcc18\src\pmc\usart\18cxx\
- OpenUSART : 開啟並設定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)

USART 的設定

- 速率：19200 , N ,8 , 1
- 資料發送：採一般模式 (polling TXIF)
- 資料接收：高優先權中斷

USART 的初始規劃

Init_MCU.C

```
void InitializeUSART(void)
{
      TRISCbits.TRISC7=1;                    // Set input for RXD
      TRISCbits.TRISC6=0;                    // Set output for TXD
      RCSTAbits.SPEN=1;                      // Enable USART Module

      OpenUSART  (  USART_TX_INT_OFF         // Set TXSTA Reg. =0b00100100
                  & USART_RX_INT_ON          // Set RCSTA Reg. =0b10010000
                  & USART_ASYNCH_MODE
                  & USART_EIGHT_BIT
                  & USART_CONT_RX
                  & USART_BRGH_HIGH
                  , 51 );                    // Set SPBRG=51, Baud Rate = 19200 @16MHz

      IPR1bits.RCIP=1;                       // Set Receive of USART are High priority
      PIE1bits.RCIE=1;                       // Enable RxD Interrupt
}
```

高優先權接收中斷函數

Main.C

```
#pragma  code  isrhighcode = 0x0008
void  isr_high_direct  (void)
{
    _asm                                    //begin in-line assembly
    goto  isr_high                          //go to isr_high function
    _endasm                                 //end in-line assembly
}
#pragma code

#pragma  interrupt  isr_high
void isr_high  (void)
{
       if (  PIR1bits.RCIF  )
       {
              Rec_Data  =  ReadUSART(  );          // Get RS-232 data
       }
}
#pragma code
```

VT100 終端機
&
指標陣列

終端機 與 GUI Application

- GUI Application
  - 圖形化的人機介面
  - 辨識及操作都非常方便
  - 須為了不同的作業系統提供不同版本的程式
  - Window 端的程式撰寫較位困難
- 終端機模擬 (VT100)
  - 大部分電腦的終端機程式都支援此種類型的終端機模擬
  - 既使是純粹的標準終端機也能支援 VT100 模式
  - 所以,使用 VT100 雖只支援文字的介面, 但相容性最高也最容易跨平台操作

使用超級終端機

- 啟動 Windows 的超級終端機
  - 在“附屬應用程式”→“通訊”→“超級終端機”
  - 設定“19200，N，8，1”；流量控制：無
  - 終端機模擬：VT100
  - 按撥號的圖示(icon)開始連線
  - 若無法順利連線則 save 至新的連線名稱後，再開啟一次

- 本實驗是使用COM1，連接 RS-232 到 APP001 的實驗板

VT100 在 WAP002 的應用

- 顯示來自RS-232的 ASCII Code 字元或字串
  - 開機時的畫面設定
  - 溫度值，PWM 輸出值
  - 各種狀態的顯示
- 利用PC的鍵盤作為資料的輸入
  - 高溫、低溫的設定值
  - 溫控系統的設定

VT-100 常用的控制命令

- 清除螢幕　　：ESC [2 J
- 清除游標右方字元：ESC [1 K
- 清除游標所在的行：ESC [2 K
- 設定游標位置　：ESC [Yn;Xn H
- 將游標移到下一行：ESC E
- 游標上移n行　　：ESC [Pn A
- 游標下移n行　　：ESC [Pn B
- 游標右移n格　　：ESC [Pn C
- 游標左移n格　　：ESC [Pn D

欲顯示的字元則以 ASCII Code 方式送出

普通型態的陣列

- 陣列是由一群具相同資料型態且相鄰位置的元素所組成的集合體
  - 格式: 資料型態 陣列名稱[n];
  - 其中n為該陣列的元素個數，實際從0到(n-1)

- 陣列中的元素表示，必須用陣列名稱 + [索引值]
  - 例如 : Array_Name[10]
  - 如欲取得陣列中某元素的位址，可用 &Array_Name[3] 的方式取得
  - 直接使用陣列名稱時，陣列名稱為一個指標
- 如果陣列的值是常數，可將該陣列設定到ROM區域

```
const rom char ch[7]={'H','e','l','l','o','\0'};
const rom char ch[]={"Hello"};
```

指標型態的陣列

- 指標型態的陣列所使到的不是陣列中的元素值，而是指到各個字串集的起始位址

VT100.C

```
const rom far char * Disp_Line[10]=
{
"                                    VT-100 Terminal",
"                Microchip Technology Taiwan ",
“          MPLAB C18 Advance Application Workshop ",
" ",
"=============================================================",
" ",
"              Temperature Monitoring & Control System ",
"              ---------------------------------------",
" ",
"    Received Slave update Counter : ",
}
```

填寫溫控系統的初始畫面

VT100.C

```
void VT100_Fill_Screen(unsigned char Line)
{
    char i;
    for (i=Line ; i<23 ; i++)                    // 顯示 0 到 23 行
    {
        putrsUSART ( Disp_Line [ i ] );
        VT100_Cursor_N_Line ( );                 // 換行命令
    }
      putrsUSART (Disp_Line [23] );              // 顯示 第 24 行

}
```

顯示溫度在 VT100 終端機

- 變數值的運算在 C 語言內是採 16 進制
- 溫度值與顯示在LCD一樣, 需經數值轉換再 VT100 顯示
  - ASCII 字元及字串可直接顯示
  - 欲顯示16進制值, 需經轉換成 ASCII
  - 欲顯示10進制值, 需經16 進制先轉換成10 進制再轉成可顯示的 ASCII

變數值為 $1000_{(16)}$ → 先轉換成 $4096_{(10)}$

$4096_{(10)}$ 再轉換成可顯示的 ASCII 字串

0x34 , 0x30 , 0x39 , 0x36 , 0x00

顯示溫度到 VT100

Main.C

```
void Print_Temperature ( int Data )
{
        if ( Data == 0 ) WriteUSART ( '0' ) ;                   // 溫度為 0, 顯示 0 度
        else
        {
                itoa ( Data,ASCII_String ) ;                    // 將溫度轉換成ASCII數字字串
                Str_Len = strlen ( ASCII_String );              // 取得ASCII數字字串的長度
                for ( i=0 ; i < Str_Len ; i++ )                 // 自動加入小數點或居先零
                {
                        if ( i == (Str_Len - 1 ) )              // 尋找小數點的位置
                        {
                                if (Str_Len==1) Print_Byte( '0' ) ;  //數字字串只有一位數, 補零
                                Print_Byte( '.' ) ;             // 加入小數點
                        }
                        Print_Byte(ASCII_String [i] ) ;         // 顯示數字字串
                }
        }
        putrsUSART ( Degree_C ) ;                               // 顯示字串 " deg. C "
}
```

練習三 關於 USART

- USART 的接收與傳送練習
  - 在 Init_MCU.C 中加入 InitializeUSART( ) 函數
    - 通信格式 = 19200 bps , 8 bit Data , No Parity , 1 Stop bit
    - 讓 USART 以中斷方式接收, 並規劃為 High Priority
      - RCIE = 1 , RCIP = 1
    - USART 的傳送中斷不要使用， 程式中將呼叫現成的函數來送出資料至 USART
  - 請記得，將原型宣告加於 Main.h

練習三 規劃 USART

Init_MCU.C

```
void InitializeUSART(void)
{
      TRISCbits.TRISC7=1;                    // Set input for RXD
      TRISCbits.TRISC6=0;                    // Set output for TXD
      RCSTAbits.SPEN=1;                      // Enable USART Module

      OpenUSART  (  USART_TX_INT_OFF         // Set TXSTA Reg. =0b00100100
                  & USART_RX_INT_ON          // Set RCSTA Reg. =0b10010000
                  & USART_ASYNCH_MODE
                  & USART_EIGHT_BIT
                  & USART_CONT_RX
                  & USART_BRGH_HIGH
                  , 51 );                    // Set SPBRG=51, Baud Rate = 19200 Bps
                                             // @ 16MHZ
      IPR1bits.RCIP=1;                       // Set Receive of USART are High priority
      PIE1bits.RCIE=1;                       // Enable RxD Interrupt
}
```

練習三的程式 Main.c

- 顯示 LCD 初始畫面
- 顯示 VT100 初始畫面
- 在 main.c 中加入以下的功能
  - 每 500 ms 就對兩個溫度 Sensor 作讀取並顯示於 LCD
  - 每次讀取將結果存於 T1_Buffer , T2_Buffer 兩變數中
  - 呼叫 VT_100_Update( ) 函數再 VT100 顯示溫度值

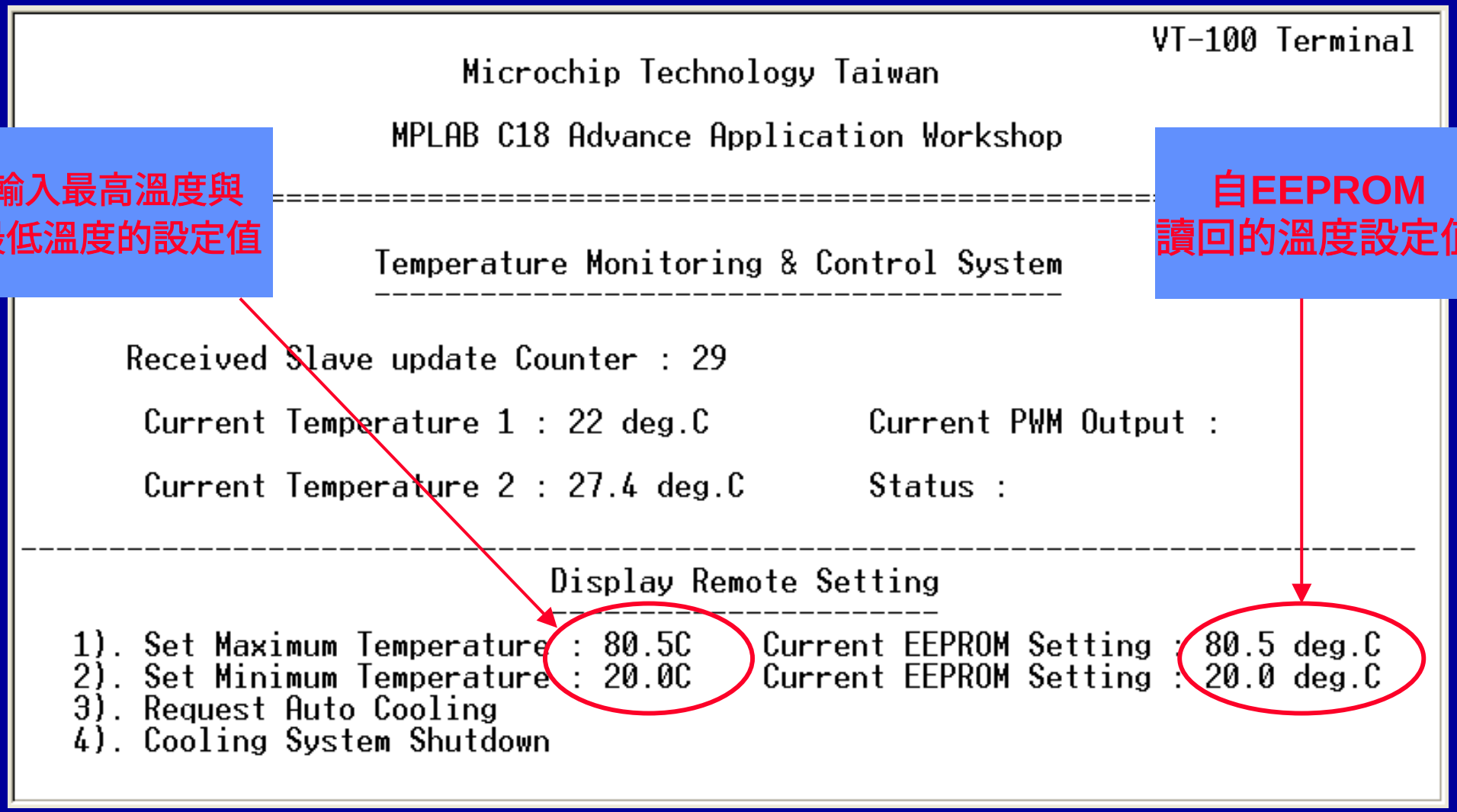
練習三

- 規劃顯示在終端機文字畫面
- 讀取兩個不同型態的溫度感應器並將其轉換後的溫度顯示在 VT100 終端機上
  - 顯示 T1 及 T2 的溫度
  - 每隔 500mS 更新顯示的溫度

第五章
存取內部 EEPROM

1. 10 進制轉 16 進制
2. 16 進制轉 10 進制
3. 存取 EEPROM

自VT100輸入溫度設定值

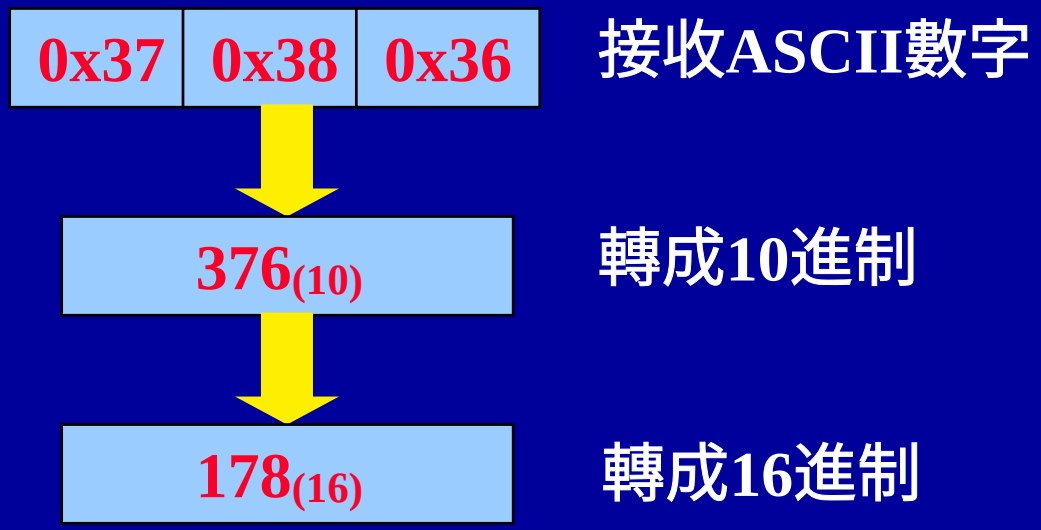


輸入最高溫度與最低溫度的設定值

自EEPROM讀回的溫度設定值

溫度的設定

- VT100 鍵盤所輸入的值為10進制的ASCII
- 程式需適當的收集這些數字後轉換成 Hex code 才能運算
  - 輸入設定溫度78.6ºC



10進制 ASCII 轉換16進制

- 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

10進制轉16進制

- C 語言
  - 將“萬位數”乘以10000 後得最大值
  - 將“千位數”乘以1000 後得第二個值
  - 將“百位數”及“十位數”做乘法轉換
  - 將所有的轉換值相加即可得一16進制值

10進制轉16進制 - 範例

將VT100輸入的溫度(000~999)轉成16進制

Rec_Cmd.c

```
near int   T2_Setting ;
near unsigned char   T2_Hi ;
near unsigned char   T2_Mid ;
near unsigned char   T2_Low ;

T2_Setting = ( int ) (T2_Hi & 0x0F) ;          // 轉換百位數, char型態轉為 int型態
T2_Setting = T2_Setting * 100 ;
T2_Mid = (T2_Mid & 0x0F) * 10 ;                // 轉換十位數
T2_Low = (T2_Low & 0x0F) ;
T2_Setting = T2_Setting + ( int ) T2_Mid ;     // "個、十、百" 三個數相加
T2_Setting = T2_Setting + ( int ) T2_Low ;
return  T2_Setting ;
```

16進制轉為10進制

- 組合語言
  - 以減法方式執行
    - 以迴圈方式連減 0x2710 直到小於 0x2710，迴圈次數就是轉換後10 進制的萬位數
    - 餘數再減 0x3E8
    - 餘數再減 0x64
    - 餘數再減 0x0A
- C 語言
  - 以 int 為例
    - 先除10000，得到整數為10進制的“萬位數”
    - 除10000 取其餘數做下一個數的運算
    - 依上步驟，再除1000，100，10

用 C 語言撰寫- 16進制轉為10進制

```
void LCD_ItoA (unsigned int AD_Data)
{
        DS_Zero_FLG = 1;
          putcLCD (Set_BCD_ASCII (AD_Data / 1000));        // 顯示千位數
          AD_Temp = AD_Temp % 1000;                        // 取出百位以後的數
          putcLCD (Set_BCD_ASCII (AD_Temp / 100));         // 顯示百位數
          AD_Temp = AD_Temp % 100;
          putcLCD (Set_BCD_ASCII (AD_Temp / 10));          // 顯示十位數
          AD_Temp = AD_Temp % 10;
          putcLCD (AD_Temp += '0');                        // 顯示個位數
}
unsigned char Set_BCD_ASCII (unsigned char BCD_Data)
{
          if (BCD_Data == 0)
          {         if (DS_Zero_FLG) return ' ';           // 居先零抑制
                    else  return '0';                      // 顯示一般的零
          }
          else
          {         DS_Zero_FLG = 0;                     // 取消居先零的抑制
                    return (BCD_Data += '0');            // 傳回相對應的 ASCII Code
          }
}
```

將16進制轉成10進制的ASCII

- btoa( )
  - 將8位元有號整數轉換成10進制的 ASCII 數字字串後存到指標指到的位址
  - 例: 0x80 -->”-128”， 100-->”100”， 199-->”-57”
- itoa( )
  - 轉換16位元的有號整數成10進制的 ASCII 數字字串
  - 例: 0x1000-->”4096”， 1000-->”1000”
- ltoa( )
  - 轉換32位元的有號整數成10進制的 ASCII 數字字串

Rec_CMD.c 的數值轉換

- Get_3_Digital ( )
  - 自 VT100 輸入三個合法10進制數字 - 即溫度設定值
  - 將溫度設定值轉成 16 進制
  - 將轉換後的16 進制溫度設定值傳給 EE_Write( ) 寫入 EEPROM 中
- EEPROM_Update ( )
  - 自 EEPROM 讀出 16 進制溫度設定值, 轉成 10 進制的 ASCII 後傳送到 VT100 顯示
- EE_Write ( )
  - 將溫度設定資料寫入 EEPROM
- EE_Read ( )
  - 自 EEPROM 讀出溫度設定資料

為何要數值轉換

- EEPROM可以儲存自VT100所輸入的 ASCII 碼也可以將存在EEPROM的ASCII碼直接送給VT100 顯示， 為何要做轉換?
  - 因為要計算溫度設定的 PWM Duty Cycle， 故須轉換成 16 進制以利於數值計算

C18的內建組合語言組譯器（一）

- 內建組合語言稱為
  - 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.h 檔案中的定義：RCIF,ADIF,W,F,FAST
- _asm 來宣告使用內建組合語言
- _endasm 作為內建組合語言的結束
- 基本常數定義是採十進制
  - movlw 10 ==> movlw 0x0a

C18的內建組合語言組譯器（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

PIC18CXXX® 簡易指令寫法

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
```

PIC18F452 內部 EEPROM 的存取

- PIC18F 系列 MCU 有內建資料 EEPROM
- 讀/寫的控制透過下列暫存器：
  - EEDATA
  - EEADR
  - EECON1
  - EECON2
    - EECON2 類似一個硬體保護鎖！必須連續被寫入 0x55，0xaa 才能啟動寫入程序
- 在寫入程序開始前務必將所有中斷暫時關閉
  - 關閉 GIEH & GIEL

PIC18F452 EEPROM 的寫入動作

```
PIR2bits.EEIF = 0;
EEADR = EE_Address;
EEDATA = EE_Data;
EECON1bits.EEPGD = 0;
EECON1bits.CFGS = 0 ;
EECON1bits.WREN = 1;
INTCONbits.GIE = 0;
_asm
        MOVLW   0X55
        MOVWF   EECON2,0
        MOVLW   0XAA
        MOVWF   EECON2,0
        BSF     EECON1,1,0
_endasm
INTCONbits.GIE = 1;
while (!PIR2bits.EEIF);
PIR2bits.EEIF = 0;
EECON1bits.WREN = 0;
```

- 寫入程序
  - 將 EEADR 與 EEDATA 寫入適當的值
  - EEPGD 位元 = 0，指向 EEPROM 記憶區
  - CFGS 位元 = 0，用來選取操作的區間為 EEPROM 及 Flash Program Memory
  - 啟動 WREN 位元
  - 關掉中斷後將硬體鎖打開
    - 0x55 > 0xaa
  - 測試 EEIF 位元即可知是否已經寫入成功了！

PIC18F452 EEPROM 的讀取

```
unsigned char EE_Read (unsigned char   EE_Address)
{
        EEADR = EE_Address;
        EECON1bits.EEPGD = 0;
        EECON1bits.CFGS = 0 ;
        EECON1bits.RD = 1;
        return EEDATA;
}
```

- 讀取程序
  - 將 EEADR 與 EEDATA 寫入適當的值
  - EEPGD 位元 = 0，指向 EEPROM 記憶區
  - CFGS 位元 = 0，用來選取操作的區間為 EEPROM 及 Flash Program Memory
  - 啟動 RD 位元
  - EEDATA 的值將可立即被讀出

練習四 的程式動作

```
while(1)
{
        Rec_Cmd_Check( ) ;

        if (Flagbits.Timer1_Flag)
        {
            Flagbits.Timer1_Flag=0;
            switch ( Timer1_Count )
               {
                        case 1 :
                        LCD_Temp_Update ( ) ;
                                    break ;
                        case 2 :
                        VT100_Update( ) ;
                                    break ;
                        case 3 :
                        EEPROM_Update( );
                                    break;
                        default :
                        break ;
                        }

                        LATDbits.LATD0 = !LATDbits.LATD0 ;
        }
}
```

- 主迴圈改為左邊的型式
  - 用 switch 來分配程式執行的時機
  - Rec_Cmd_Check( ) 會處理由終端機接收的相關資訊
  - VT100_Update 會將溫度值顯示在 (13,32) & (15,32)
  - EEPROM_Update 會將 EEPROM 內的設定值顯示在 (20,68) & (21,68)

練習四

- 將VT100終端機輸入的溫度設定值轉換成16 進制值後存入 EEPROM 中
- 讀取 EEPROM 的溫度設定值轉換成10 進制值後顯示到 VT100 的螢幕上
- 將 PIC18F452 燒成 Stand-Alone Mode 後關閉電源在開機，以測試溫度值是否存入 EEPROM 中

第六章
PWM 的計算

1. 變數視野
2. PWM 模組
3. Duty Cycle 計算

變數視野的擴展

- 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
- 對於是外部 (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 變數的傳遞

- 使用參數的傳遞
  - void VT100_puthex (unsigned char HEX_Val)
  - 呼叫 VT100_puthex 時，必須將一參數傳入
  - 此參數的傳遞一般是透過軟體堆疊或額外的 RAM (static local 被 enable 時)
- 可使用公用變數來取代參數列的傳遞，比較省時間且增加執行速度, 但會佔用較多記憶位址
- 變數在該程式宣告，只有該程式可以看得到，其它的程式並無法使用該變數
- 如程式中需使用其他程式所宣告的變數，可以用extern 來擴展視野

使用 extern

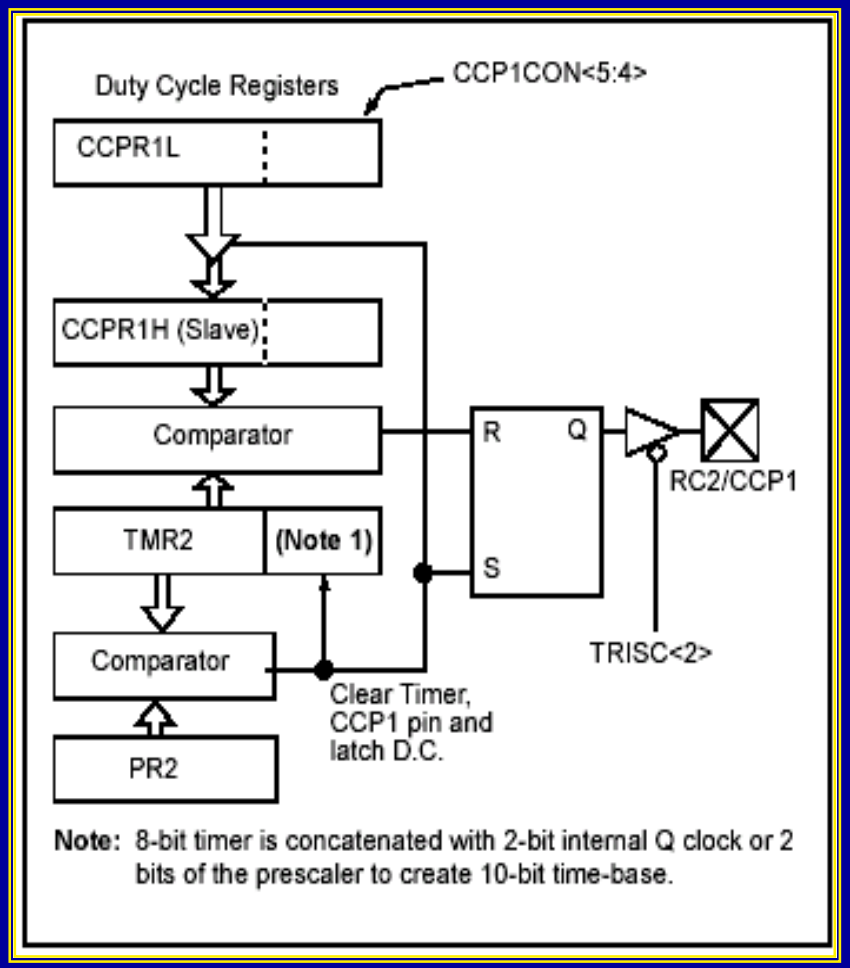
Main.C

```
near unsigned char Rec_Data;
near unsigned char Buzzer_Count;
near union
{
        unsigned char Flag;
        struct {
                unsigned Key_Flag:1;
                unsigned Timer1_Flag:1;
                unsigned Pause_Flag:1;
                unsigned Buzzer_On_Flag:1;
                unsigned Buzzer_Fast_Flag:1;
                unsigned Buzzer_Mid_Flag:1;
                unsigned Buzzer_Slow_Flag:1;
                unsigned Shutdown_Flag:1;
                };
} Flagbits;
```

Rec_Cmd.C

```
extern near unsigned char Rec_Data;
extern near unsigned char Buzzer_Count;
extern near union
{
        unsigned char Flag;
        struct {
                unsigned Key_Flag:1;
                unsigned Timer1_Flag:1;
                unsigned Pause_Flag:1;
                unsigned Buzzer_On_Flag:1;
                unsigned Buzzer_Fast_Flag:1;
                unsigned Buzzer_Mid_Flag:1;
                unsigned Buzzer_Slow_Flag:1;
                unsigned Shutdown_Flag:1;
                };
} Flagbits;
```

使用 PWM 來作輸出控制



- TIMER 2 為 PWM 的計時暫存器
  - 與 PR2 相比較來決定 Period
  - 與 CCPRxH 比較來決定 PWM 的 Duty
  - TMR2 = PR2 時:
    - TMR2 會自動歸零, 並將 CCPx 設為 1(High)
    - 若 Duty 為 0 , 則 CCPx 維持 0 (Low)
    - CCPRxL , CCPxCON<5:4> 將會被載入 CCPRxH 及 2 位元隱含位元組成的 10 bit 暫存器
  - 當 TMR2 = CCPRxH 時, CCPx 會被設定為 0 (Low)

PWM 動作的詳細說明

- 由 PIC18F 的規格來看 PWM 是 10 bit 的解析度, 但是 …
  - TMR2 , PR2 , CCPRxH , CCPRxL 皆為 8 bit 的暫存器 !!!
- PWM 的 Duty 由 CCPRxL 及 CCPxCON 的 bit 4:5 組成 10 bits 的設定值
- CCPRxH 其實在內部與附加的兩個位元共組成 10 bit 的空間來存放由 CCPRxL , CCPxCON<4:5> 來的設定值
- TMR2 雖只有 8 bit , 但其他兩個 bit 來自於
  - 若 Timer2 的 Prescaler 為 1:1
    - 2 bit 的 Internal Q Clock (每個指令由 4 個 Q Clock 組成 , 記得嗎 ?)
  - 若 Timer2 的 Prescaler 為 1:4 or 1:16
    - 2 bit 的預除器 (Pre-scaler)

Timer2 的初始化 !!

```
void InitializeTMR2(void)
{
        OpenTimer2 (TIMER_INT_OFF
                & T2_PS_1_4
                &T2_POST_1_1);
        PR2 = 0xFF;

}
```

- Timer2 有 Pre-scaler 與 Post-Scaler
- Timer2 的 Prescaler 影響到 Timer2 Reset 的時間與 (Period)
- 若中斷有被 Enable , 則 Timer2 的 Post-Scaler 設定值將影響 Timer2 中斷 CPU 的頻率

PWM 的 Initial !!

- 在對 Timer2 做完初始化工作後才對 PWM 做初始化的工作 !!
- 可使用既有的函數 OpenPWM1(period) , OpenPWM2(period)
- PWM 週期 = (period+1) *4 * Tosc * TMR2 Prescaler
- TMR2 的 Prescaler 設為 1:4 , 故得到的週期為 256 uS = 3.9K Hz
- 3.9K Hz 為人類聽覺範圍, 用它來同時供應 Buzzer 及 FAN 的控制

```
void InitializePWM(void)
{
        OpenPWM1(0xFF);          // Open PWM1 for Buzzer
        OpenPWM2(0xFF);          // OPen PWM2 for FAN
        SetDCPWM1(0x3FF/2);      // Buzzer out frequency
                                 // PWM = (PR2+1)*4*(1/16MHz)(Prescal)
                                 // 3.9KKz = 256 *4*0.0625uS * 4 = 256 uS
        TRISCbits.TRISC2=1;      // Turn off  the PWM1 output      (Buzzer)
        TRISBbits.TRISB3=0;      // Turn on the PWM2 output        (FAN)
}
```

PWM 輸出值的計算

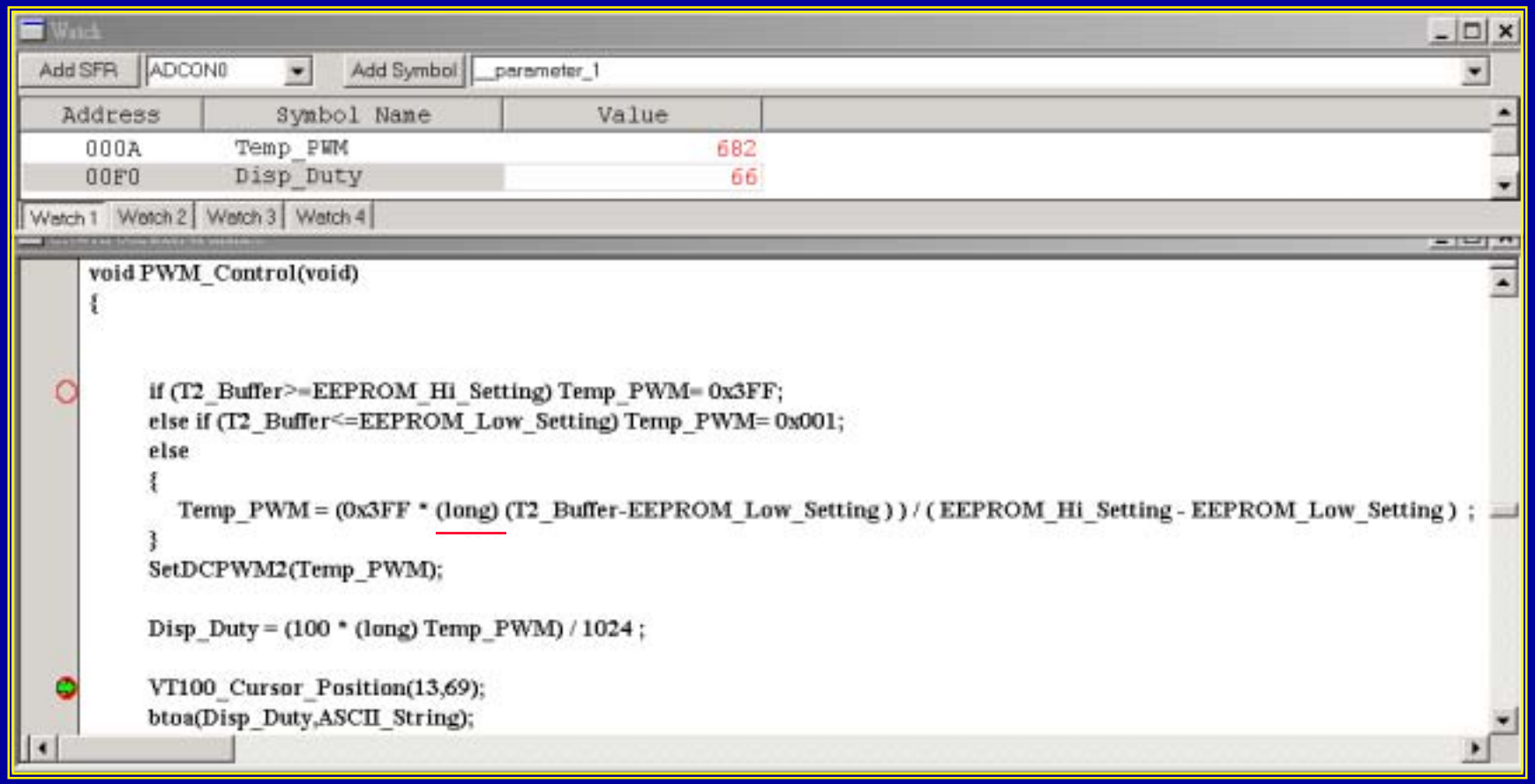
- 溫度的最高與最低容許值分別被設定於下列兩變數中
  - EEPROM_Hi-Setting , EEPROM_Low_Setting
- T2 的溫度值被存放於 T2_Buffer 變數
- 若 PWM 的 Duty Cycle 為 0 .. 1023 則應使用以下的公式來計算要輸出的 Duty Cycle

$$\left\{ \frac{(T2\_Buffer - EEPROM\_Low\_Setting) * 1024}{(EEPROM\_High\_Setting - EEPROM\_Low\text{-}Setting)} \right\} = Temp\_PWM$$

- 使用 SetDCPWM2(Temp_PWM-1) 來設定 Duty Cycle
- 相同的 , 知道了實際的 PWM Duty 後可利用 Temp_PWM 來計算要顯示的輸出百分比 (Disp_Duty)
  - Disp_Duty = [(Temp_PWM) / 1024] * 100

PWM 輸出值的計算

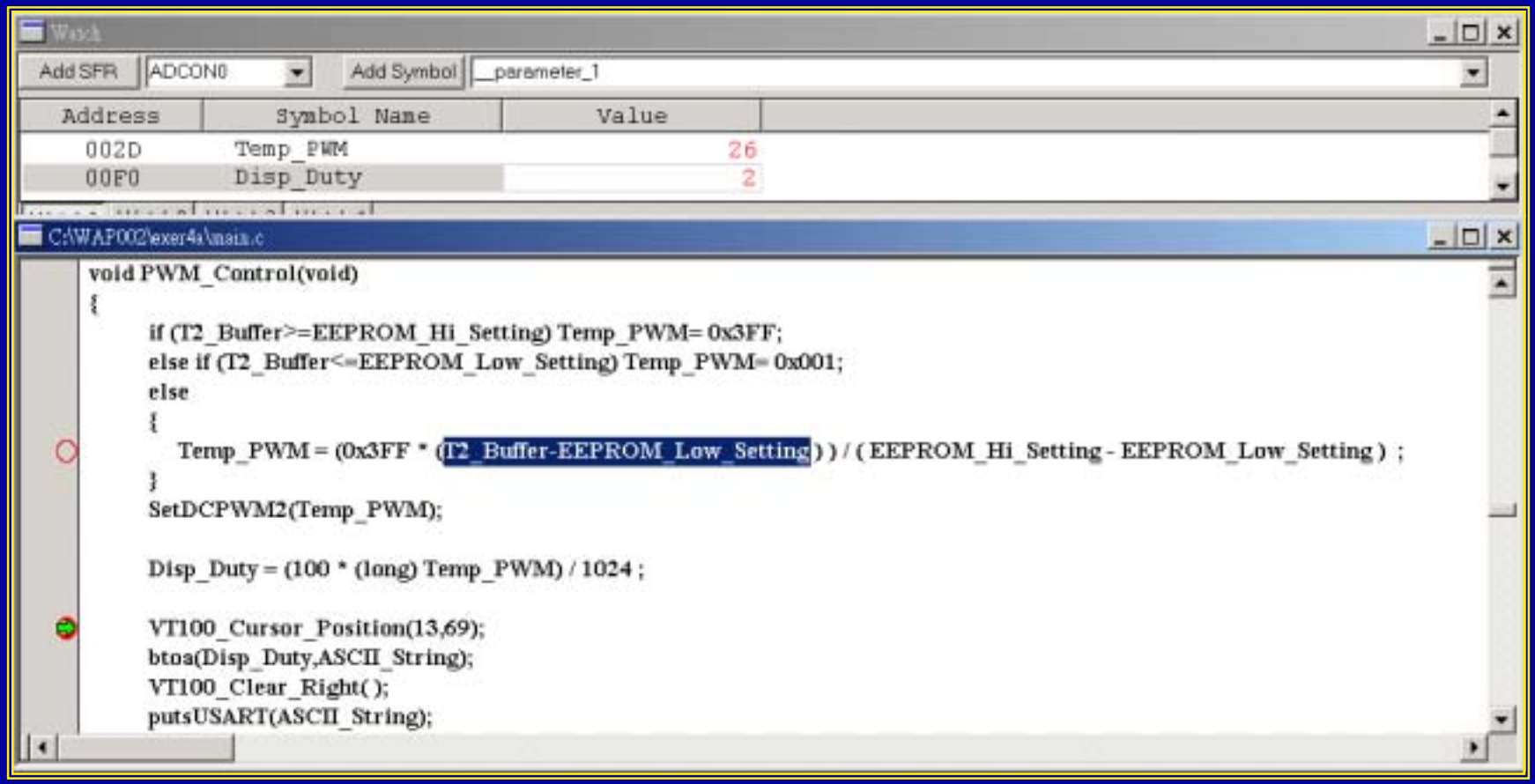
- 假設 EEPROM_Hi_Setting = 800 , EEPROM_Low_Setting = 200 , T2_Buffer = 600
- 依照公式計算 , Temp_PWM = 682 , Disp-Duty = 66 是正確的值



PWM 輸出值的計算

- 把計算 Temp_PWM 數值式中的 (long) 改型運算子拿掉後結果就錯了！

為什麼 ??



MPLAB C18 的改型運算子

- 在計算 Temp_PWM 的式子中, 最先被計算的是
  - (1024 * (T2-Buffer – EEPROM_Low_Setting))
- 請注意 , 1024 * 400 = 409600
- 409600 遠超過 int 型別能容納的大小 (0 .. 65535)
  - 這是為何計算的結果出錯的原因 !!
- 使用 (long) 改型運算子
  - 將計算的結果使用 “long” 的資料型態來儲存並繼續未完成的運算
- !! 請用改型運算子來避免計算時因溢位產生的軟體過失

PWM 輸出值的計算

```
void PWM_Control(void)
{
        if (T2_Buffer>=EEPROM_Hi_Setting) Temp_PWM= 0x3FF;
        else if (T2_Buffer<=EEPROM_Low_Setting) Temp_PWM= 0x001;
        else
        {
        Temp_PWM = (1024 * (long) (T2_Buffer-EEPROM_Low_Setting ) ) /
                        (EEPROM_Hi_Setting - EEPROM_Low_Setting )  ;
        }
        SetDCPWM2(Temp_PWM – 1 );

        Disp_Duty = (100 * (long) Temp_PWM) / 1024 ;

        VT100_Cursor_Position(13,69);
        btoa(Disp_Duty,ASCII_String);
        VT100_Clear_Right( );
        putsUSART(ASCII_String);
        Print_Byte('%');
}
```

// 請注意防止溢位所需的改型運算 !!

加入 PWM_Control

- 在 main( ) 的主迴圈加上可執行到 PWM_Control( ) 的程式，請注意要檢查 Timer 1 中斷中對 Timer1_Count 的更新是否正確 !!

```
while(1)
{
        Rec_Cmd_Check( ) ;
        if (Flagbits.Timer1_Flag)
        {
                Flagbits.Timer1_Flag=0;
                switch ( Timer1_Count )
                {
                        case 0 :        LCD_Temp_Update ( ) ;
                                        break ;
                        case 1 :        VT100_Update( ) ;
                                        break ;
                        case 2 :        EEPROM_Update( );
                                        break;
                        case 3 :        PWM_Control( ) ;
                                        break ;
                        default :       break ;
                }
        }
}
```

按鍵彈跳問題

```
if (SW2_Debance!=0)
      {
      if (!SW2 | !SW3) SW2_Debance=10;
      else  SW2_Debance--;
      }
      else
      {
            if (!SW3)
            {
            Flagbits.Key_Flag =1 ;
            SW2_Debance =10;
            }
            if (!SW2)
            {
            Flagbits.SW2_Flag = !Flagbits.SW2_Flag;
            SW2_Debance =10;
            }
      }
```

- 按鍵開關均有彈跳問題，如不處理易造成程式的誤判
- 處理彈跳須利用連續檢查方式處理
- 以軟體作為計時方式，複雜且易浪費 MCU 資源
- 以 Timer 中斷方式處理最具有效率
- 以 5mS 中斷連續檢查 10 次，共 50 mS
- 如果按鍵接點太差，可將檢驗次數加長

練習五

- PWM_Control ( )
  - 計算目前 T2 的溫度與 EEPROM 所設定的溫度點間的 PWM Duty Cycle 輸出， 並將其結果顯示至 VT100
  - 利用 PWM 模式驅動風扇並以Duty Cycle方式控制轉速
- 在 Read_Tmp.c 中加入 SW2 的控制， 可利用 VR 輸入模擬的溫度值
- 利用 Timer2 來處理按鍵彈跳問題

第七章

完成溫度控制系統

WAP-002 智慧型警報系統需求

O.K. ● 量測兩個溫度感應器，並顯示目前溫度在 LCD 及 VT-100 終端機

O.K. ● User 可透過 VT-100 終端機的鍵盤來設定最高、最低溫度的控制點並存入 溫度控制站的 EEPROM

O.K. ● User 可透過 VT-100 終端機開啟或關閉溫度控制站的控制模式

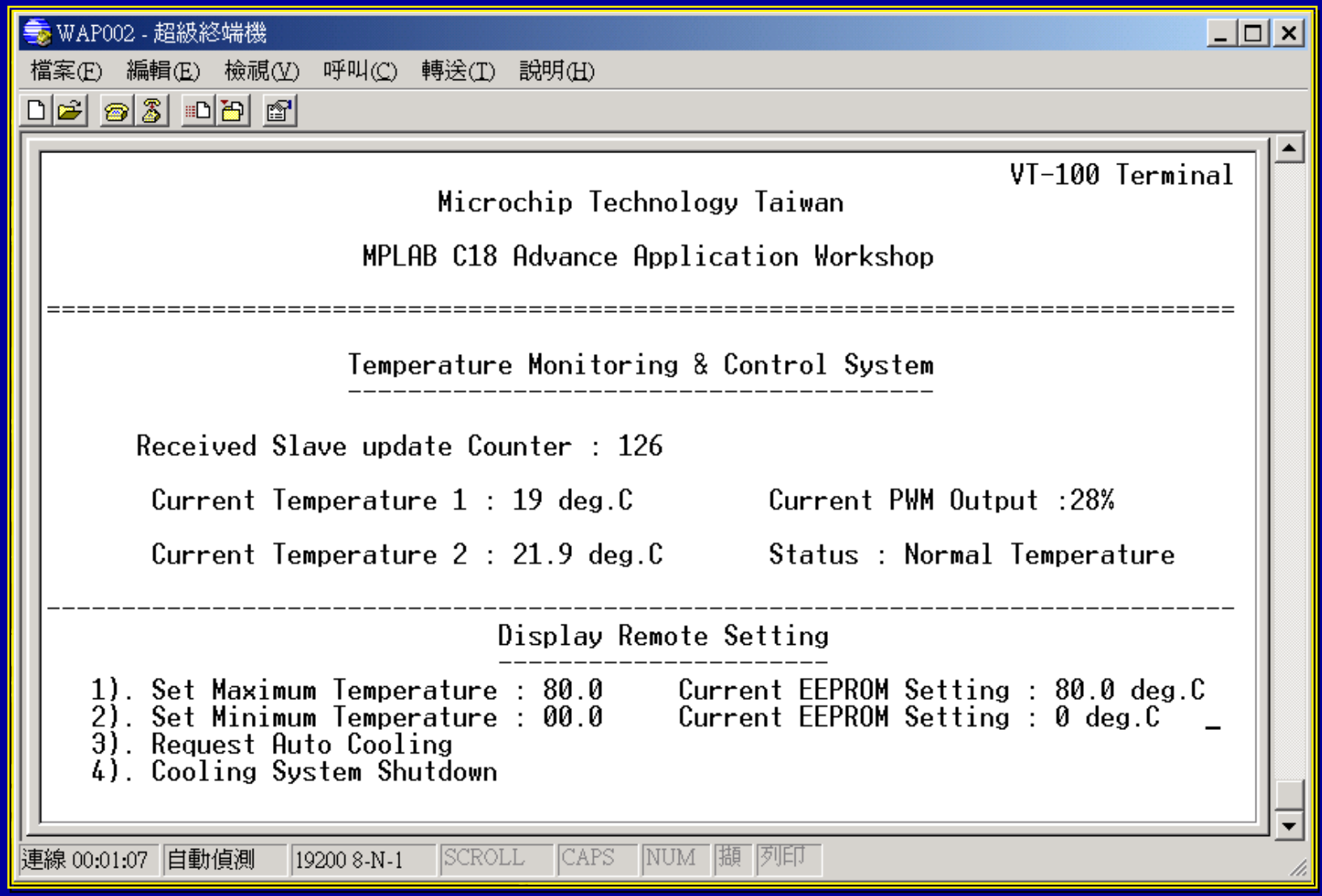
O.K. ● 溫度警報系統的輸出為一個 10-bit PWM 輸出，其輸出 (Duty) 會隨 EEPROM 的最高、最低溫度設定點與目前所測得的溫度計算出 Duty 的輸出值 (1% to 99%)，以控制散熱風扇的轉速

● 溫度警報系統有一蜂鳴器，會隨溫度發出不同的警報聲響

- ◆ 溫度超過 98% -- 連續警報音
- ◆ 溫度介於兩點間的 75% - 97% -- 段續警報音
- ◆ 溫度介於兩點間的 1% - 74% -- 關閉警報音
- ◆ 溫度低 1% -- 段續長聲警報音

剩下這個部分就完成

最終的成果 – 智慧溫控警報系統



為何要使用 分時多工 的技巧

- 程式架構一目了然
- 各個程式皆可獨立，除錯或修改較簡單
- 適合寫較大的程式，系統擴充容易
- 可依程式的功能需求，分配執行時間

分時多工程式

- 將程式的流程確定後， 建議以分時多工的技巧來完成程式
- 利用Timer為分配的時間來源
  - PIC18xxxx 內有豐富的 Timer 資源
- 確定每個 Task (分離的作業程式) 沒有永久迴圈 或 Dead Loop
  - 如果 EEPROM 與 CPU 的通信無法正確，必須有判斷錯誤與逾時的機制存在 !!
- 確定每個 Task 的執行時間小於分配時間

Main.c 的主程式

```
while(1)                                                    // 主程式的迴圈
{
        Rec_Cmd_Check( );                                   // 即時檢查RS-232 接收程式
        Key_Press_Check( );                                 // 即時檢查按鍵是否按下

        if (Flagbits.Timer1_Flag)                           // Timer1 的分時是否被設定(0.1s)
        {
                Flagbits.Timer1_Flag=0;                     // 是的，將分時旗號清為零

                switch (Timer1_Count)                       // 檢查分時計數器，輪到誰做
                {
                        case 0x00:
                        if (LCD_Count==0) LCD_Temp_Update( );   // 每0.5s update 溫度顯示到LCD
                        break;

                        case 0x01:
                        VT_100_Update( );                   // 每0.5s update VT100的顯示資料
                        break;

                        case 0x02:
                        EEPROM_Update( );                   // 每0.5s 讀取EEPROM內的溫度設定
                        break;

                        case 0x03:
                        PWM_Control( );                     // 計算溫度與PWM的 Duty Cycle
                        break;

                        case 0x04:
                        Temp_Compare( );                    // 目前溫度與EEPROM的設定溫度做比較
                        break;
```

Temp_Compare 的動作要求

- Temp_Compare( ) 以 Disp_Duty 為判斷條件
  - 若 Disp_Duty >=99
    - 顯示 Over Temperature 於座標 (15,58)
    - 設定警報音為高速 (Flagbits.Buzzer_Fast_Flag=1)
  - 若 Disp_Duty >= 75 , Disp_Duty < 99
    - 顯示 High Temperature 於座標 (15,58)
    - 設定警報音為中速 (Flagbits.Buzzer_Mid_Flag=1)
  - 若 Disp_Duty >= 25 , Disp_Duty < 75
    - 顯示 Normal Temperature 於座標 (15,58) 並關掉警報音
  - 若 Disp_Duty >= 1 , Disp_Duty < 25
    - 顯示 Lower Temperature 於座標 (15,58) 並關掉警報音
  - 若 Disp_Duty < 1
    - 顯示 Lost Temperature 於座標 (15,58)
    - 設定警報音為低速 (Flagbits.Buzzer_Slow_Flag=1)

控制警報聲響速度的技巧

- 在 Timer1 的 ISR 中加入以下的片段來控制 Buzzer

```
if (Buzzer_Count==0) TRISCbits.TRISC2=1;                    //  Buzzer control , If the count=0, uzzer off
else
{
        Buzzer_Count--;
        Bz.Count++;
        if (Flagbits.Buzzer_Fast_Flag==1)                   // Check the alarm with fast mode
                {
                if (Bz.B1==1) TRISCbits.TRISC2=0;           // Turn On the PWM1 output (Buzzer)
                else TRISCbits.TRISC2=1;                    // Turn off the PWM1 output (Buzzer)
                }
        else if (Flagbits.Buzzer_Mid_Flag==1)               // Check the alarm with slow mode
                {
                if (Bz.B2==1) TRISCbits.TRISC2=0;           // Turn On the PWM1 output (Buzzer)
                else TRISCbits.TRISC2=1;                    // Turn off the PWM1 output (Buzzer)
                }
        else if (Flagbits.Buzzer_Slow_Flag==1)              // Check the alarm with slow mode
                {
                if (Bz.B4==1) TRISCbits.TRISC2=0;           // Turn On the PWM1 output (Buzzer)
                else TRISCbits.TRISC2=1;                    // Turn off the PWM1 output (Buzzer)
                }
}
```

控制警報聲響速度的技巧

```
near union
{
   unsigned char Count;
   struct {
            unsigned B0:1;
            unsigned B1:1;
            unsigned B2:1;
            unsigned B3:1;
            unsigned B4:1;
            unsigned B5:1;
         };
} Bz=0;
```

- 使用一個名為 Bz 的 union 變數
- 配合不同的旗標來選擇速度
  - Flagbits.Buzzer_Fast_Flag
  - Flagbits.Buzzer_Mid_Flag
  - Flagbits.Buzzer_Slow_Flag
- 每個旗標檢查在 Bz 的不同的位元

```
if (Flagbits.Buzzer_Fast_Flag==1)
{
if (Bz.B1==1)  TRISCbits.TRISC2=0;      // Turn On the PWM1 output
else   TRISCbits.TRISC2=1;              // Turn off the PWM1 output
}
else if (Flagbits.Buzzer_Mid_Flag==1)
{
if (Bz.B2==1)  TRISCbits.TRISC2=0;      // Turn On the PWM1 output
else   TRISCbits.TRISC2=1;              // Turn off the PWM1 output
}
```

- 因為 B1 與 B2 在不同位置，故產生的聲音長度也不同

練習六 完成溫控系統

- 將 PWM 的輸出加以量化成成五種百分比的輸出
  - 溫度介於 99% -100% -- 連續警報音（顯示 Over Temperature）
  - 溫度介於 75% - 98% -- 段續警報音（顯示 High Temperature）
  - 溫度介於 25% - 74% -- 關閉警報音（顯示 Normal Temperature）
  - 溫度介於 1% - 24% -- 關閉警報音（顯示 Lower Temperature）
  - 溫度介於 0% 以下 -- 段續長聲警報音（顯示 Lost Temperature）
- 利用 Bz.Count 計數器各位元不同倍數時間的特性來產生 BUZZER 的長短聲
- 程式完成後，程式碼總長為 7134 bytes，在 PIC18Fxxx 中只佔不到 4KW 的空間