TÀI LIỆU SỬ DỤNG CCS TIẾNG VIỆT

I / _ GIỚI THIỆU:

_Đây là tài liệu hướng dẫn sử dụng CCS lập trình ngôn ngữ C cho vi điều khiển PIC của Microchip . Tác giả tên **TRẦN XUÂN TRƯỜNG** , SV K2001 ,ĐH BK HCM . Là thành viên **txt2203** trên diễn đàn **diendandientu.com .** Mọi đóng góp ý kiến về tài liệu xin vào mục **Vi xử lý-Vi điều khiển** của diễn đàn hoặc email đến địa chỉ : txt2203@yahoo.com . Rất cám ơn mọi đóng góp ý kiến của các bạn yêu thích lập trình vi xử lý đối với tài liệu này .

II / _ VÀI VẤN ĐỀ VỀ TÀI LIỆU NÀY :

_Tài liệu hướng dẫn sử dụng phần mềm CCS các phiên bản , lập trình C cho VĐK . Tài liệu cũng giải thích cách thức hoạt động của 1 số module của VĐK để các bạn nắm rõ hơn hoạt động VĐK nhằm sử dụng hàm và viết chương trình 1 cách chính xác .
_Tài liệu này không chủ ý thay thế hoàn toàn HELP của CCS , nó chỉ là phần cô đọng , là hướng dẫn viết 1 chương trình bắt đầu từ đâu , giới thiệu 1 số hàm và cách hoạt động , 1 số vấn đề khi lập trình , . . . do đó nó không đầy đủ , bạn nên đối chiếu tài liệu này với HELP tiếng Anh để nắm rõ vấn đề , đồng thời học cả tiếng Anh dễ dàng .
_CCS có phần " common questions " – những câu hỏi thường gặp và trả lời , chưa được dịch ở đây dù nó rất quan trọng , nhiều bạn hay bỏ qua , không để ý . Bạn nên xem nó .
_Tài liệu trình bày về các vấn đề sau :

_ Chương 0 : Giới thiệu sơ lược CCS . Viết 1 chương trình C trong CCS như thế nào . Công cụ mô phỏng .
_ Chương 1 : Sử dụng biến và hàm , các cấu trúc lệnh , chỉ thị tiền xử lý.
_ Chương 2 : Các hàm xử lý số , xử lý bit , delay .
_ Chương 3 : Xử lý ADC , các hàm vào , ra trong C .
_ Chương 4 : Truyền thông với PC , xử lý chuỗi .
_ Chương 5 : vấn đề TIMER.
_ Chương 6 : Truyền thông I2C , SPI và PARALLEL .
_ Chương 7 : Các vấn đề về PWM , Capture và Compare .
_ Chương 8 : Vấn đề ngắt (interrupt).

_Trong mỗi chương sẽ có các hướng dẫn sử dụng các hàm thích hợp cho chủ đề đó .
_Nên đọc chương 1 trước . Các chương còn lại độc lập nhau . Thích gì đọc nấy .
_Tài liệu này viết đến đâu đưa lên mạng cho mọi người tham khảo đến đó . Hầu hết sẽ trình bày sử dụng 1 cách cơ bản nhất , sau đó sẽ bổ sung thêm VD,chương trình, . . . thêm đến đâu sẽ thông báo đến đó .

CHƯƠNG 0 :

HƯỚNG DẪN SỬ DỤNG CCS – VIẾT CHƯƠNG TRÌNH C TRONG CCS

I / _ GIỚI THIỆU CCS :

_Chương trình CCS dùng cho tài liệu này là PCW COMPILER version 3.07 (2001) hoặc 3.222 (2004) , bao gồm : PCB , PCM và PCH . Phiên bản mới nhất là 3.227 có nhiều hàm mới và chức năng mới , cập nhật mới . Lập trình cho các họ PIC 12 bit , 14 bit và PIC 18 .

_Để viết 1 chương trình C mới : chạy CCS , vào New để tạo 1 file C mới . Trên thanh toolbar :

_Chọn "Microchip 12 bit" để viết chương trình cho PIC 12 bit . "Microchip 14 bit" để viết chương trình cho PIC 14 bit . "Microchip PIC18" để viết chương trình cho PIC18 .

_Chọn "Compiler" để biên dịch chương trình bạn đang viết.

_CCS là trình biên dịch dùng ngôn ngữ C lập trình cho VĐK . Đây là ngôn ngữ lập trình đầy sức mạnh , giúp bạn nhanh chóng trong việc viết chương trình hơn so với ngôn ngữ Assembly .

_Tuy nhiên C không phải là vạn năng , có thể thực hiện mọi thứ như ý muốn . Trong 1 số trường hợp , nó có thể sinh mã chạy sai (tham khảo các cải tiến ở các version CCS trên web : info.CCS.com Mặt khác , nó sinh mã không theo ý muốn (dù không sai , ví dụ như sinh nhiều mã lệnh không quan trọng khi thực thi hàm ngắt) làm chậm tốc độ thực thi chương trình nếu bạn đòi hỏi chương trình xử lý với tốc độ cao , ví dụ như điều chế PWM .

_Nhưng CCS C cho phép bạn phối hợp ASSEMBLY cùnh với C , điều này cho phép chương trình của bạn sẽ trở nên rất uyển chuyển , kết hợp được sức mạnh của cả 2 ngôn ngữ , dù rằng việc phối hợp sẽ làm cho việc viết chương trình trở nên khó khăn hơn .

_CCS cung cấp các công cụ tiện ích giám sát hoạt động chương trình như : C/ASM list : cho phép xem mã ASM của file bạn biên dịch , giúp bạn quản lý mã và nắm được các thức mã sinh ra và nó chạy như thế nào , là công cụ rất quan trọng , bạn có thể gỡ rối chương trình và nắm được hoạt động của nó ; SYMBOL hiển thị bộ nhớ cấp phát cho từng biến , giúp quản lý bộ nhớ các biến chương trình ,. . . CallTree hiển thị phân bổ bộ nhớ .

_ Có nhiều tiện ích trong mục Tools , nhưng do bản crack nên nhiều cái không xài được .

II / _ CÔNG CỤ MÔ PHỎNG , TÍCH HỢP TRONG MPLAB :

_Công cụ mô phỏng cho PIC 16Fxxx. . đa năng nhất chỉ có thể là PIC Simulator IDE 5x , hỗ trợ 38 loại PIC 16Fxxx . Có cả Oscilocope , INT ảo , . . . và nhiều chức năng khác với giao diện tuyệt đẹp , dễ dùng . Hoạt động độc lập , lấy file HEX để mô phỏng . Có dịch ngược ra Assemble . Có bộ lập trình BASIC và Assemble rất hay và dễ dùng , dù khá đơn giản nhưng đủ để viết các chương trình nhỏ chất lượng . Bạn nên thử qua .

_ Mô phỏng với PIC 18 , PIC 12 , và nhiều loại PIC 16 mà IDE trên không hỗ trợ ? Bạn có thể dùng CCS tích hợp MPLAB .

III / _ CCS TÍCH HỢP TRONG MPLAB :

_Bạn có thể soạn mã CCS trong môi trường MPLAB và mô phỏng mã C (không phải Assemble) , tương tự như lập trình và mô phỏng với MPLAB C18 .

_ Thiết lập môi trường CCS : vào MPLAB IDE , vô mục Project-> Set Language Tool Location . . . Hộp thoại mở ra , nhấn vào dấu + của dòng CCS C Compile ->Executable ,sau đó chọn Browser để thiết lập đường dẫn đến file ccsc . exe trong thư mục cài đặt CCS (tên là PICC) . Bạn có thể thêm đường dẫn vào 4 dòng của Default Search Path . . . nếu thấy cần .

_ Tạo 1 dự án (project) CCS trong MPLAB : vào Project-> Project Wizard , chọn VĐK làm việc , ở bước 2 : chọn bộ công cụ (Active toolSuite) là CCS C Compiler . . . có thể không cần Add file thiết bị *.h vì trong file mã mà bạn sẽ viết sau đó có dòng #include file này rồi thì nó tự include vào thôi . thế là bạn đã có môi trường làm việc CCS trong MLPAB . Khuyết điểm là bạn không thể dùng các tiện ích của CCS độc lập được (C/asm list . . .) . Tuy vậy , bạn có thể soạn mã từ CCS độc lập , ném qua MPLAB , để dùng được tính năng mô phỏng C của MPLAB .

_ Để mô phỏng : sau khi soạn mã , chọn Compile . Bạn có thể dùng mọi tiện ích trong mục View để mô phỏng . Mở Watch , chọn các biến C mà bạn muốn quan sát , thanh ghi đặc biệt muốn xem . Sau đó mở Debugger->Select tool->MPLAB SIM . Tool bar mô phỏng xuất hiện , chọn animate để chạy từng dòng lệnh C mô phỏng .

_ Lưu ý : vào Debugger-> Setting . . . để thay đổi các thiết lập cần thiết : OSC/TRACK : thay đổi tần số VĐK thích hợp . ANIMATION / REALTIME UPDATE : để thay đổi tốc độ mô phỏng và cập nhật Watch . Mục này còn dùng để thay đổi tốc độ mô phỏng cho file mã Assemble (mặc định nó chạy như RÙA ấy – 1 s cho 1 lệnh) .

II / _ VIẾT 1 CHƯƠNG TRÌNH TRONG CCS :

_Sau đây là ví dụ 1 chương trình trong CCS :

```
#include < 16F877 .h >
#device PIC6f877 *=16 ADC=10
#use delay(clock=20000000)
 . . . .
Int16 a,b;
 . . . .
Void xu_ly_ADC ( )
{ . . .
```

```
 . . .
}

#INT_TIMER1
Void xu_ly_ngat_timer ( )
{ . . .
 . . .
}

Main ( )
{ . . .
 . . .
}
```

_Đầu tiên là các chỉ thị tiền xử lý : # . . . có nhiệm vụ báo cho CCS cần sử dụng những gì trong chương trình C như dùng VXL gì , có dùng giao tiếp PC không , ADC không , DELAY không , . . .
_Các khai báo biến .
_Các hàm con .
_ Các hàm phục vụ ngắt theo sau bởi 1 chỉ thị tiền xử lý cho biết dùng ngắt nào.
_Chương trình chính .

CHƯƠNG 1 :

CÁCH SỬ DỤNG BIẾN VÀ HÀM , CÁC CẤU TRÚC LỆNH , CHỈ THỊ TIỀN XỬ LÝ

I / _ KHAI BÁO VÀ SỬ DỤNG BIẾN , HẰNG , MẢNG :

1 / _ Khai báo biến , hằng ,mảng :

_Các loại biến sau được hỗ trợ :

int1	số 1 bit = true hay false (0 hay 1)
int8	số nguyên 1 byte (8 bit)
int16	số nguyên 16 bit
int32	số nguyên 32 bit
char	ký tự 8 bit
float	số thực 32 bit
short	mặc định như kiểu int1
byte	mặc định như kiểu int8
int	mặc định như kiểu int8
long	mặc định như kiểu int16

_Thêm signed hoặc unsigned phía trước để chỉ đó là số có dấu hay không dấu .Khai báo như trên mặc định là không dấu . 4 khai báo cuối không nên dùng vì dễ nhầm lẫn . Thay vào đó nên dùng 4 khai báo đầu .

VD :
Signed int8 a ; // số a là 8 bit dấu (bit 7 là bit dấu).
Signed int16 b , c , d ;
Signed int32 , . . .
_Phạm vi biến :
Int8 :0 , 255 signed int8 : -128 , 127
Int16 : 0 ,2^15-1 signed int16 : -2^15 , 2^15-1
Int32 : 0 , 2^32-1 signed int32 : -2^31 , 2^31-1

_Khai báo hằng : VD :
Int8 const a=231 ;
_Khai báo 1 mảng hằng số :
VD : Int8 const a[5] = { 3,5,6,8,6 } ; //5 phần tử , chỉ số mảng bắt đầu từ 0 : a[0]=3
_Một mảng hằng số có kích thước tối đa tuỳ thuộc loại VĐK:
*NếuVĐK là PIC 14 (VD :16F877) : bạn chỉ được khai báo 1 mảng hằng số có kích thước tối đa là 256 byte .
Các khai báo sau là hợp lệ :
Int8 const a[5]={ . . .}; // sử dụng 5 byte , dấu . . . để bạn điền số vào
Int8 const a[256]={ . . .}; // 256 phần tử x 1 byte = 256 byte
Int16 const a[12] = { . . . }; // 12 x 2= 24 byte
Int16 const a[128] = { . . . }; // 128 x 2= 256 byte
Int16 const a[200] = { . . . }; // 200 x 2 =400 byte : không hợp lệ
*Nếu VĐK là PIC 18 : khai báo mảng hằng số thoải mái , không giới hạn kích thước .
_Lưu ý : nếu đánh không đủ số phần tử vào trong ngoặc kép như đã khai báo , các phần tử còn lại sẽ là 0 . Truy xuất giá trị vượt quá chỉ số mảng khai báo sẽ làm chương trình chạy vô tận .
_Mảng hằng số thường dùng làm bảng tra (ví dụ bảng tra sin) , viết dễ dàng và nhanh chóng , gọn hơn so với khi dùng ASM để viết .
_Khai báo 1 biến mảng : kích thước tuỳ thuộc khai báo con trỏ trong #device và loại VDK:
*PIC 14 : Nếu bạn khai báo con trỏ 8 bit : VD # device *=8 : không gian bộ nhớ chỉ có 256 byte cho tất cả các biến chương trình bất chấp VĐK của bạn có hơn 256 byte RAM (Vd : 368 , . . .) và biến mảng có kích thước tối đa tuỳ thuộc độ phân mảnh bộ nhớ , với 16F877 có 368 byte ram , thường thì kích thước không quá 60 byte ,có khi dưới 40 byte , nếu khai báo lớn hơn sẽ gặp lỗi vô duyên : not enough ram for all variable trong khi thực sự VDK còn rất nhiều RAM . Nếu khai báo con trỏ 16 bit : VD : #device *=16 , không gian bộ nhớ là đầy đủ (trừ đi 1 ít RAM do CCS chiếm làm biến tạm) .VD : với 16F877 bạn dùng đủ 368 byte RAM . Nhưng kích thước mảng cũng không quá 60 byte .
* PIC 18 : kích thước mảng không giới hạn, xài hết RAM thì thôi . Với khai báo con trỏ 8 bit , bạn chỉ được xài tối đa 256 byte RAM , nếu khai báo con trỏ 16 bit , bạn xài trọn bộ nhớ RAM thực sự .
_VD Khai báo biến mảng : int16 a[125] ; // biến mảng 126 phần tử , kích thước 252 byte ram .

2 / _ Cách sử dụng biến :

_Khi sử dụng các phép toán cần lưu ý : sự tràn số , tính toán với số âm , sự chuyển kiểu và ép kiểu .

A) _Một vài ví dụ về tràn số , làm tròn :

_VD :

Int8 a=275; // a =275-256=19
Int8 const a=275 //a=19
Int8 a=40 , b=7 , c;
C=a * b ; //c=280-256=24
C=a / b ; //c=5

_Bạn có thể ép kiểu , thường là tiết kiệm ram , hay muốn tiết kiệm thời gian tính , ... VD :
Int8 a =8 , b=200;
Int16 c ;
C= (int16) a * b ;
// c= 1600 , a chuyển sang 16 bit , 16bit*8bit→ b tự động chuyển sang 16 bit , kết quả là 16 bit trong c , lưu ý biến a , b vẫn là 8 bit .
_8bit * 8bit → phép nhân là 8 bit , KQ là 8 bit
_16bit * 8 bit → phép nhân là 16 bit , KQ là 16 bit
_32bit * 16 bit → phép nhân là 32 bit , KQ là 32 bit
_16bit * 16 bit → phép nhân là 16 bit , KQ là 16 bit
. . . v . v . . .
_Có thể ép kiểu kết quả : VD : 16b*8b→16bit , nếu gán vào biến 8 bit thì KQ sẽ cắt bỏ 8 bit cao .

II / _CÁC CẤU TRÚC LỆNH : (statement)

_Gồm các lệnh như while . . do , case , . . .

STATEMENTS

STATEMENT	EXAMPLE

if (expr) **stmt; [else stmt;]**	`if (x==25)`
	`    x=1;`
	`else`
	`    x=x+1;`

while **(expr) stmt;**	`while (get_rtcc()!=0)` `   putc('n');`

do **stmt** while **(expr);**	`do {`
	`   putc(c=getc());`
	`} while (c!=0);`

for **(expr1;expr2;expr3) stmt;**	`for (i=1;i<=10;++i)` `   printf("%u\r\n",i);`

switch (expr) {
case **cexpr: stmt; //one or more case**
[default:**stmt]**
... }
	`switch (cmd) {`
	`   case 0:  printf("cmd 0");`
	`      break;`
	`   case 1: printf("cmd 1");`
	`      break;`
	`   default: printf("bad cmd");`
	`      break; }`

return **[expr];**	`return (5);`
goto **label;**	`goto loop;`
label: stmt;	`loop: I++;`
break;	`break;`
continue;	`continue;`
expr;	`i=1;`

;	;
{[stmt]} Zero or more	{a=1; b=1;}

Lưu ý : các mục trong [ ] là có thể có hoặc không .

_while **(expr) stmt :** xét điều kiện trước rồi thực thi biểu thức sau .
_ do **stmt** while **(expr) :** thực thi biểu thức rồi mới xét điều kiện sau .
_**Return** : dùng cho hàm có trả về trị , hoặc không trả về trị cũng được , khi đó chỉ cần dùng: return ; (nghĩa là thoát khỏi hàm tại đó) .
_**Break** : ngắt ngang (thoát khỏi) vòng lặp while. _**Continue** : quay trở về đầu vòng lặp while .

III / _ CHỈ THỊ TIỀN XỬ LÝ :

_Xem chi tiết tất cả ở phần HELP , mục pre_processor . Ở đây sẽ giới thiệu 1 số chỉ thị thường dùng nhất :

1 / _ #ASM và #ENDASM :

_Cho phép đặt 1 đoạn mã ASM giữa 2 chỉ thị này , Chỉ đặt trong hàm . CCS định nghĩa sẵn 1 biến 8 bit _RETURN_ để bạn gán giá trị trả về cho hàm từ đoạn mã Assembly.
_C đủ mạnh để thay thế Assmemly . Vì vậy nên hạn chế lồng mã Assembly vào vì thường gây ra xáo trộn dẫn đến sau khi biên dịch mã chạy sai , trừ phi bạn nắm rõ Assembly và đọc hiểu mã Assembly sinh ra thông qua mục C/Asm list .
_Khi sử dụng các biến không ở bank hiện tại , CCS sinh thêm mã chuyển bank tự động cho các biến đó . Nếu sử dụng #ASM ASIS thì CCS không sinh thêm mã chuyển bank tự động , bạn phải tự thêm vào trong mã ASM .
_Lưu ý : mã Assembly theo đúng mã tập lệnh VDK , không phải mã kiểu MPLAB .
_VD :

```
int find_parity (int data)    {
int count;
#asm
movlw   0x8
movwf   count
movlw   0
loop:
xorwf   data,w
rrf     data,f
decfsz  count,f
goto    loop
movwf   _return_
#endasm
}
```

2 / _ #INCLUDE :

_Cú pháp : #include <filename>
Hay #include “ filename”
Filename : tên file cho thiết bị *.h , *.c . Nếu chỉ định file ở đường dẫn khác thì thêm đường dẫn vào . Luôn phải có để khai báo chương trình viết cho VĐK nào , và luôn đặt ở dòng đầu tiên .
_VD :

```
#include  <16F877.H>                    // chương trình sử dụng cho VĐK 16F877

#include < C:\INCLUDES\COMLIB\MYRS232.C >
```

3 / _ #BIT , #BYTE , #LOCATE và # DEFINE:

_ #BIT id = x . y

Với id : tên biến x : biến C (8,16,32,...bit) hay hằng số địa chỉ thanh ghi.

y : vị trí bit trong x

→ tạo biến 1 bit đặt ở byte x vị trí bit y, tiện dùng kiểm tra hay gán trị cho bit thanh ghi . Điểm khác biệt so với dùng biến 1 bit từ khai báo int1 là : int1 tốn 1 bit bộ nhớ , đặt ở thanh ghi đa mục đích nào đó do CCS tự chọn , còn #BIT thì không tốn thêm bộ nhớ do id chỉ là danh định đại diện cho bit chỉ định ở biến x , thay đổi giá trị id (0 / 1) sẽ thay đổi giá trị bit tương ứng y -> thay đổi trị x.

_VD:

#bit TMR1Flag = 0xb.2 //bit cờ ngắt timer1 ở địa chỉ 0xb.2 (PIC16F877)

Khi đó TMR1Flag = 0 → xoá cờ ngắt timer1

Int16 a=35; //a=00000000 00100011

#bit b= a.11 //b=0 , nếu b=a.0 thì b chỉ vị trí LSB (bit thấp nhất , bên trái)

Sau đó : b=1; //a=00001000 00100011 = 2083

_Lưu ý không dùng được : if (0xb.2) mà phải khai báo như trên rồi dùng : if(TMR1Flag)

_#BYTE id = x

X: địa chỉ id : tên biến C

Gán tên biến id cho địa chỉ (thanh ghi) x , sau đó muốn gán hay kiểm tra địa chỉ x chỉ cần dùng id . Không tốn thêm bộ nhớ , tên id thường dùng tên gợi nhớ chức năng thanh ghi ở địa chỉ đó . Lưu ý rằng giá trị thanh ghi có thể thay đổi bất kỳ lúc nào do hoạt động chương trình nên giá trị id cũng tự thay đổi theo giá trị thanh ghi đó . Không nên dùng id cho thanh ghi đa mục đích như 1 cách dùng biến int8 vì CCS có thể dùng các thanh ghi này bất kỳ lúc nào cho chương trình , nếu muốn dùng riêng , hãy dùng #LOCATE.

_VD:

#byte port_b = 0xc6; // 16F877 :0xc6 là địa chỉ portb

Muốn port b có giá trị 120 thì : port_b=120;

#byte status = 0xc3;

_ # LOCATE id = x

_Làm việc như #byte nhưng có thêm chức năng bảo vệ không cho CCS sử dụng địa chỉ đó vào mục đích khác . VD: # LOCATE temp = 0xc20 // 0xc20 :thanh ghi đa mục đích

Cách sau tương tự :

Int8 temp ;

#locate temp = 0xc20

_ Sử dụng #LOCATE để gán biến cho 1 dãy địa chỉ kề nhau (cặp thanh ghi) sẽ tiện lợi hơn thay vì phải dùng 2 biến với #byte .

VD : CCP1 có giá trị là cặp thanh ghi 0x15 (byte thấp) và 0x16 (byte cao) . Để gán trị cho CCP1 :

Int16 CCP1;

#locate CCP1= 0x15 // byte thấp của CCP1 ở 0x15 , byte cao của CCP1 ở 0x16

Gán trị cho CCP1 sẽ tự động gán vào cả 2 thanh ghi

CCP1 = 1133 ; // = 00000100 01101101 → 0x15 = 00000100 , 0x16 = 01101101

_# DEFINE id text

Text : chuỗi hay số . Dùng định nghĩa giá trị .

VD : #define a 12345

4 / _ # DEVICE :

DEVICE chip option
chip : tên VĐK sử dụng , không dùng tham số này nếu đã khai báo tên chip ở # include .
option : toán tử tiêu chuẩn theo từng chip:

* = 5 dùng pointer 5 bit (tất cả PIC)
* = 8 dùng pointer 8 bit (PIC14 và PIC18)
* = 16 dùng pointer 16 bit (PIC14 ,PIC 18)
ADC = x sử dụng ADC x bit (8 , 10 , . . . bit tuỳ chip) , khi dùng hàm read_adc( ) , sẽ trả về giá trị x bit .
ICD = true : tạo mã tương thích debug phần cứng Microchip
HIGH_INTS = TRUE : cho phép dùng ngắt ưu tiên cao

_Khai báo pointer 8 bit , bạn sử dụng được tối đa 256 byte RAM cho tất cả biến chương trình .
_Khai báo pointer 16 bit , bạn sử dụng được hết số RAM có của VDK .
_Chỉ nên dùng duy nhất 1 khai báo #device cho cả pointer và ADC .
VD : #device * = 16 ADC = 10

5 / _ # ORG :

\# org ***start , end***
\# org ***segment***
\#org ***start , end { }***

Start , end: bắt đầu và kết thúc vùng ROM dành riêng cho hàm theo sau , hoặc để riêng không dùng .

VD :

```
Org 0x30 , 0x1F
Void xu_ly( )
{
}        // hàm này bắt đầu ở địa chỉ 0x30

org 0x1E00
anotherfunc( )
{
}        //hàm này bắt đầu tuỳ ý ở 0x1E00 đến 0x1F00

Org 0x30 , 0x1F { }
         // không có gì cả đặt trong vùng ROM này
```

_Thường thì không dùng ORG .

6 / _ # USE :

 # USE delay (clock = speed)
Speed : giá trị OSC mà bạn dùng . VD: dùng thạch anh dao động 40Mhz thì :
#use delay(clock = 40000000)
_Chỉ khi có chỉ thị này thì trong chương trình bạn mới được dùng hàm delay_us ( ) và delay_ms( ) .

#USE fast_io (port**)**
Port : là tên port :từ A-G (tuỳ chip)

_Dùng cái này thì trong chương trình khi dùng các lệnh io như output_low() , . . . nó sẽ set chỉ với 1 lệnh , nhanh hơn so với khi không dùng chỉ thị này.
_Trong hàm main( ) bạn phải dùng hàm set_tris_x( ) để chỉ rõ chân vào ra thì chỉ thị trên mới có hiệu lực , không thì chương trình sẽ chạy sai .
_Không cần dùng nếu không có yêu cầu gì đặc biệt .
VD : # use fast_io(A)

#USE I2C (options)
_Thiết lập giao tiếp I2C.
Option bao gồm các thông số sau, cách nhau bởi dấu phẩy :

Master	: chip ở chế độ master
Slave	: chip ở chế độ slave
SCL = pin	: chỉ định chân SCL
SDA = pin	: chỉ định chân SDA
ADDRESS =x	: chỉ định địa chỉ chế độ slave
FAST	: chỉ định FAST I2C
SLOW	: chỉ định SLOW I2C
RESTART_WDT	: restart WDT trong khi chờ I2C_READ( )
FORCE_HW	: sử dụng chúc năng phần cứng I2C (nếu chip hỗ trợ)
NOFLOAT_HIGH	: không cho phép tín hiệu ở float high (???) , tín hiệu được lái từ thấp lên cao.
SMBUS	: bus dùng không phải bus I2C , nhưng là cái gì đó tương tự .

_VD :
#use I2C (master , sda=pin_B0 , scl = pin_B1)
#use I2C (slave , sda= pin_C4 , scl= pin_C3 , address = 0xa00 , FORCE_HW)

#USE RS232 (options)
_Thiết lập giao tiếp RS232 cho chip (có hiệu lực sau khi nạp chương trình cho chip , không phải giao tiếp RS232 đang sử dụng để nạp chip) .
Option bao gồm :

BAUD = x	: thiết lập tốc độ baud rate : 19200 , 38400 , 9600 , . . .
PARITY = x	: x= N ,E hay O , với N : không dùng bit chẵn lẻ .
XMIT = pin	: set chân transmit (chuyển data)
RCV = pin	: set chân receive (nhận data)

_Các thông số trên hay dùng nhất , các tham số khác sẽ bổ sung sau.
VD :
#use rs232(baud=19200,parity=n,xmit=pin_C6,rcv=pin_C7)

7 / Một số chỉ thị tiền xử lý khác :

#CASE : cho phép phân biệt chữ hoa / thường trong tên biến , dành cho những ai quen lập trình C .

#OPT n :với n=0 – 9 : chỉ định cấp độ tối ưu mã , không cần dùng thì mặc định là 9 (very tối ưu) .

#PRIORITY ints : với ints là danh sách các ngắt theo thứ tự ưu tiên thực hiện khi có nhiều ngắt xảy ra đồng thời , ngắt đứng đầu sẽ là ngắt ưu tiên nhất , dùng ngắt nào đưa ngắt đó vô . Chỉ cần dùng nếu dùng hơn 1 ngắt . Xem cụ thể phần ngắt .
VD : #priority int_CCP1 , int_timer1 // ngắt CCP1 ưu tiên nhất

MỘT SỐ VẤN ĐỀ QUAN TRỌNG KHÁC – xem chi tiết trong phần HELP :

_Biểu thức : xem HELP->Expressions , trong đó : biểu thị số trong C:

123 : số decimal 0x3 , 0xB1 : số hex 0b100110 : số binary

'a' : ký tự

"abcd" : chuỗi , ký tự null được thêm phía sau

_Các toán tử C : xem Operators

>= , < = , = = , != (không bằng)

&& : and || : or ! : not (đảo của bit , không phải đảo của byte)

>>n : dịch trái n bit << n : dịch phải n bit

++ , - - , += , - = , . . .

CHƯƠNG 2 :

CÁC HÀM XỬ LÝ SỐ , XỬ LÝ BIT , DELAY

I / CÁC HÀM XỬ LÝ SỐ :

_Bao gồm các hàm:

Sin() cos() tan() Asin() acos() atan()

Abs() : lấy trị tuyệt đối

Ceil( ) :làm tròn theo hướng tăng

Floor ( ) :làm tròn theo hướng giảm

Exp ( ) : tính e^x

Log ( ) :

Log10 ( ) :

Pow ( ) : tính luỹ thừa

Sqrt ( ) :căn thức

_Các hàm này chạy rất chậm trên các VDK không có bộ nhân phần cứng (PIC 14 ,12) vì chủ yếu tính toán với số thực và trả về cũng số thực (32 bit) và bằng phần mềm .VD hàm sin mất 3.5 ms (thạch anh = 20Mhz)để cho KQ . Do đó nếu không đòi hỏi tốc độ thì dùng các hàm này cho đơn giản , như là dùng hàm sin thì khỏi phải lập bảng tra.

_Xem chi tiết trên HELP CCS , cũng dễ đọc thôi mà. Hơn nữa chúng ít dùng .

II / CÁC HÀM XỬ LÝ BIT VÀ CÁC PHÉP TOÁN :

_Bao gồm các hàmsau :

Shift_right() shift_left()

Rotate_right() rotate_left()

Bit_clear() bit_set() bit_test() Swap()

Make8() make16() make32()

1 / Shift_right (address , byte , value)
Shift_left (address , byte , value)

_Dịch phải (trái) 1 bit vào 1 mảng hay 1 cấu trúc . Địa chỉ có thể là địa chỉ mảng hay địa chỉ trỏ tới cấu trúc (kiểu như &data) . Bit 0 byte thấp nhất là LSB .

2 / Rotate_right () , rotate_left ()

_Nói chung 4 hàm này ít sử dụng .

3 / Bit_clear (var , bit)
Bit_set (var , bit)

_Bit_clear ( ) dùng xóa (set = 0) bit được chỉ định bởi vị trí **bit** trong biến **var** .
_Bit_set ( ) dùng set=1 bit được chỉ định bởi vị trí **bit** trong biến **var** .
_var : biến 8 , 16 , 32 bit bất kỳ .
_bit : vị trí clear (set) : từ 0-7 (biến 8 bit) , 0-15 (biến 16 bit) , 0-31 (biến 32 bit) .
_Hàm không trả về trị .
VD :
Int x;
X=11 ;   //x=1011
Bit_clear (x ,1) ;   // x= 1001b = 9

4 / Bit_test (var , bit) :

_Dùng kiểm tra vị trí **bit** trong biến **var** .
_Hàm trả về 0 hay 1 là giá trị bit đó trong **var** .
_var : biến 8, 16 ,32 bit .
_bit : vị trí bit trong **var .**
_Giả sử bạn có biến x 32 bit đếm từ 0 lên và muốn kiểm tra xem nó có lớn hơn 4096 không (4096= 2^12 =1000000000000b) :
If (x >= 4096) . . .   // phép kiểm tra này mất ~5 us
Trong 1 vòng lặp , việc kiểm tra thường xuyên như vậy sẽ làm mất 1 thời gian đáng kể . Để tối ưu , chỉ cần dùng : if (bit_test (x, 12) → chỉ mất ~ 0.4 us . (20 Mhz thạch anh) .
_Kiểm tra đếm lên tới những giá trị đặc biệt (2^ i) thì dùng hàm này rất tiện lợi.

5 / Swap (var) :

_var : biến 1 byte
_Hàm này tráo vị trí 4 bit trên với 4 bit dưới của var , tương đương var =(var>>4) | (var << 4)
_Hàm không trả về trị .
VD :
X= 5 ;   //x=00000101b
Swap (x) ;   //x = 01010000b = 80

6 / make8 (var , offset) :

_Hàm này trích 1 byte từ biến var .
_var : biến 8,16,32 bit . offset là vị trí của byte cần trích (0,1,2,3) .
_Hàm trả về giá trị byte cần trích .
VD :
Int16 x = 1453 ;   // x=0x5AD
Y = Make(x, 1) ;   //Y= 5 = 0x05

7 / make16 (varhigh , varlow) :

_Trả về giá trị 16 bit kết hợp từ 2 biến 8 bit varhigh và varlow . Byte cao là varhigh , thấp là varlow .

8 / make32 (var1 , var2 , var3 , var4) :

_Trả về giá trị 32 bit kết hợp từ các giá trị 8 bit hay 16 bit từ var1 tới var4 . Trong đó var2 đến var4 có thể có hoặc không . Giá trị var1 sẽ là MSB , kế tiếp là var2 , . . .Nếu tổng số bit kết hợp ít hơn 32 bit thì 0 được thêm vào MSB cho đủ 32 bit .

VD:

```
Int a=0x01 , b=0x02 , c=0x03 , d=0x04 ;          // các giá trị hex
Int32 e ;
e = make32 (  a , b , c , d );              // e = 0x01020304
e = make32 ( a , b , c , 5 ) ;              // e = 0x01020305
e = make32 ( a, b, 8 );                     // e = 0x00010208
e = make32 ( a ,0x1237 ) ;                  // e = 0x00011237
```

III / CÁC HÀM DELAY :

_Để sử dụng các hàm delay , cần có khai báo tiền xử lý ở đầu file , VD : sử dụng OSC 20 Mhz , bạn cần khai báo : #use delay (clock = 20000000)

_Hàm delay không sử dụng bất kỳ timer nào . Chúng thực ra là 1 nhóm lệnh ASM để khi thực thi từ đầu tới cuối thì xong khoảng thời gian mà bạn quy định . Tuỳ thời gian delay yêu cầu dài ngắn mà CCS sinh mã phù hợp . có khi là vài lệnh NOP cho thời gian rất nhỏ . Hay 1 vòng lặp NOP . Hoặc gọi tới 1 hàm phức tạp trong trường hợp delay dài . Các lệnh nói chung là vớ vẩn sao cho đủ thời gian quy định là được . Nếu trong trong thời gian delay lại xảy ra ngắt thì thời gian thực thi ngắt không tính vào thời gian delay , xong ngắt nó quay về chạy tiếp các dòng mã cho tới khi xong hàm delay . Do đó thời gian delay sẽ không đúng .

_Có 3 hàm phục vụ :

1 / delay_cycles (count)

Count : hằng số từ 0 – 255 , là số chu kỳ lệnh .1 chu kỳ lệnh bằng 4 chu kỳ máy .

_Hàm không trả về trị . Hàm dùng delay 1 số chu kỳ lệnh cho trước .

VD : delay_cycles (25) ;   // với OSC = 20 Mhz , hàm này delay 5 us

2 / delay_us (time)

Time : là biến số thì = 0 – 255 , time là 1 hằng số thì = 0 -65535 .

_Hàm không trả về trị .

_Hàm này cho phép delay khoảng thời gian dài hơn theo đơn vị us .

_Quan sát trong C / asm list bạn sẽ thấy với time dài ngắn khác nhau , CSS sinh mã khác nhau .

3 / delay_ms (time)

Time = 0-255 nếu là biến số hay = 0-65535 nếu là hằng số .

_Hàm không trả về trị .

_ Hàm này cho phép delay dài hơn nữa .

VD :

```
Int a = 215;
Delay_us ( a ) ;          // delay 215 us
Delay_us ( 4356 ) ;     // delay 4356 us
Delay_ms ( 2500 ) ;    // delay 2 . 5 s
```

CHƯƠNG 3 :

XỬ LÝ ADC , CÁC HÀM I / O TRONG C

I / _ XỬ LÝ ADC :

_PIC có nhiều chân phục vụ xử lý ADC với nhiều cách thức khác nhau . Để dùng ADC , bạn phải có khai báo #DEVICE cho biết dùng ADC mấy bit (tuỳ chip hỗ trợ , thường là 8 hay 10 bit hoặc hơn) . Bạn cần lưu ý là: 1 VDK hỗ trợ ADC 10 bit thì giá trị vào luôn là 10 bit , nhưng chia cho 4 thì còn 8 bit . Do đó 1 biến trở chiết áp cấp cho ngõ vào ADC mà bạn chọn chế độ 10 bit thì sẽ rất nhạy so với chế độ 8 bit (vì 2 bit cuối có thay đổi cũng không ảnh hưởng giá trị 8 bit cao và do đó kết quả 8 bit ADC ít thay đổi) , nếu chương trình có chế độ kiểm tra ADC để cập nhật tính toán , hay dùng ngắt ADC , thì nó sẽ chạy hoài thôi . Dùng ADC 8 bit sẽ hạn chế điều này . Do đó mà CCS cung cấp chọn lựa ADC 8 hay 10 bit tùy mục đích sử dụng .

Cấu hình bộ ADC :

_Thông dụng nhất khi dùng ADC là sử dụng 1 biến trở , điều chỉnh bởi 1 nút vặn , qua đó thu được 1 điện áp nhỏ hơn điện áp tham chiếu (Vref – áp max) , đưa vào chân biến đổi ADC , kết quả cho 1 giá trị số ADC 8 bit (0-255) hay ADC 10 bit (0-1023) . Thường thì áp Vref lấy bằng Vdd (5V) (??)
_Trên các PIC có ngõ AVdd và AVss (PIC 18) , thường thì bạn luôn nối AVdd tới Vdd , AVss tới Vss để đảm bảo hoạt động cho lập trình qua ICD 2 .
Các hàm sau phục vụ ADC :

1 / Setup_ADC (**mode**) :

_Không trả về trị . Dùng xác định cách thức hoạt động bộ biến đổi ADC . Tham số mode tuỳ thuộc file thiết bị *.h có tên tương ứng tên chip bạn đang dùng , nằm trong thư mục DEVICES của CCS . Muốn biết có bao nhiêu tham số có thể dùng cho chip đó , bạn mở file tương ứng đọc , tìm tới chỗ các định nghĩa cho chức năng ADC dùng cho chip đó tương ứng với hàm này . Sau đây là các giá trị mode của 16F877 , (1 số khác có thể không có hoặc có thêm như 16F877A có thêm 1 số thứ là ADC_CLOCK_DIV_2/4/8/16/32/64 . . .) :

ADC_OFF : tắt hoạt động ADC (tiết kiệm điện , dành chân cho hoạt động khác) .
ADC_CLOCK_INTERNAL : thời gian lấy mẫu bằng xung clock IC (mất 2-6 us) thường là chung cho các chip .
ADC_CLOCK_DIV_2 : thời gian lấy mẫu bằng xung clock / 2 (mất 0.4 us trên thạch anh 20MHz)
ADC_CLOCK_DIV_8 : thời gian lấy mẫu bằng xung clock / 8 (1.6 us)
ADC_CLOCK_DIV_32 : thời gian lấy mẫu bằng xung clock / 32 (6.4 us)

2 / Setup_ADC_ports (**value**)

_Xác định chân lấy tín hiệu analog và điện thế chuẩn sử dụng . Tùy thuộc bố trí chân trên chip , số chân và chân nào dùng cho ADC và số chức năng ADC mỗi chip mà value có thể có những giá trị khác nhau. Xem file tương ứng trong thư mục DEVICES để biết số chức năng tương ứng chip đó .
Để tương thích chương trình viết cho phiên bản cũ , 1 số tham số có 2 tên khác nhau (nhưng cùng

chức năng do định nghĩa cùng địa chỉ) , ở đây dùng phiên bản 3.227 .Lưu ý : Vref : áp chuẩn , Vdd : áp nguồn

Sau đây là các giá trị cho value (chỉ dùng 1 trong các giá trị) của 16F877 :
ALL_ANALOGS : dùng tất cả chân sau làm analog : A0 A1 A2 A3 A5 E0 E1 E2 (Vref=Vdd)
NO_ANALOG : không dùng analog , các chân đó sẽ là chân I /O .
AN0_AN1_AN2_AN4_AN5_AN6_AN7_VSS_VREF : A0 A1 A2 A5 E0 E1 E2 VRefh=A3
AN0_AN1_AN2_AN3_AN4 : A0 A1 A2 A3 A5
(tên thì giống nhau cho tất cả thiết bị nhưng 16F877 chỉ có portA có 5 chân nên A0 , A1 , A2 , A5 được dùng , A6 , A7 không có)
AN0_AN1_AN3 : A0 A1 A3 , Vref = Vdd
AN0_AN1_VSS_VREF : A0 A1 VRefh = A3
AN0_AN1_AN4_AN5_AN6_AN7_VREF_VREF : A0 A1 A5 E0 E1 E2 VRefh=A3 , VRefl=A2 .
AN0_AN1_AN2_AN3_AN4_AN5 : A0 A1 A2 A3 A5 E0
AN0_AN1_AN2_AN4_AN5_VSS_VREF : A0 A1 A2 A5 E0 VRefh=A3
AN0_AN1_AN4_AN5_VREF_VREF : A0 A1 A5 E0 VRefh=A3 VRefl=A2
AN0_AN1_AN4_VREF_VREF : A0 A1 A5 VRefh=A3 VRefl=A2
AN0_AN1_VREF_VREF : A0 A1 VRefh=A3 VRefl=A2
AN0 : A0
AN0_VREF_VREF : A0 VRefh=A3 VRefl=A2

VD : setup_adc_ports (AN0_AN1_AN3) ; // A0 , A1 , A3 nhận analog , áp nguồn +5V cấp cho IC sẽ là điện áp chuẩn .

3 / Set_ADC_channel (**channel**) :

_Chọn chân để đọc vào giá trị analog bằng lệnh Read_ADC ( ) . Giá trị channel tuỳ số chân chức năng ADC mỗi chip .Với 16F877 , channel có giá trị từ 0 -7 :
0-chân A0 1-chân A1 2-chân A2 3-chân A3 4-chân A5
5-chân E0 6-chân E1 7-chân E2
_Hàm không trả về trị . Nên delay 10 us sau hàm này rồi mới dùng hàm read_ADC ( ) để bảo đảm kết quả đúng . Hàm chỉ hoạt động với A /D phần cứng trên chip.

4 / Read_ADC (**mode**) :

_Dùng đọc giá trị ADC từ thanh ghi (/ cặp thanh ghi) chứa kết quả biến đổi ADC . Lưu ý hàm này sẽ hỏi vòng cờ cho tới khi cờ này báo đã hoàn thành biến đổi ADC (sẽ mất vài us) thì xong hàm .
_Nếu giá trị ADC là 8 bit như khai báo trong chỉ thị #DEVICE , giá trị trả về của hàm là 8 bit , ngược lại là 16 bit nếu khai báo #DEVICE sử dụng ADC 10 bit trở lên .
_Khi dùng hàm này , nó sẽ lấy ADC từ chân bạn chọn trong hàm Set_ADC_channel( ) trước đó . Nghĩa là mỗi lần chỉ đọc 1 kênh Muốn đổi sang đọc chân nào , dùng hàm set_ADC_channel( ) lấy chân đó . Nếu không có đổi chân , dùng read_ADC( ) bao nhiêu lần cũng được .

_mode có thể có hoặc không , gồm có :
ADC_START_AND_READ : giá trị mặc định
ADC_START_ONLY : bắt đầu chuyển đổi và trả về
ADC_READ_ONLY : đọc kết quả chuyển đổi lần cuối

#DEVCE	8 bit	10 bit	11 bit	16 bit
ADC=8	0-255	0-255	00-255	00-255
ADC=10	x	0-1023	x	x
ADC=11	x	x	0-2047	x
ADC=16	0-65280	0-65472	0-65504	0-65535

_16F877 chỉ hỗ trợ ADC 8 và 10 bit .

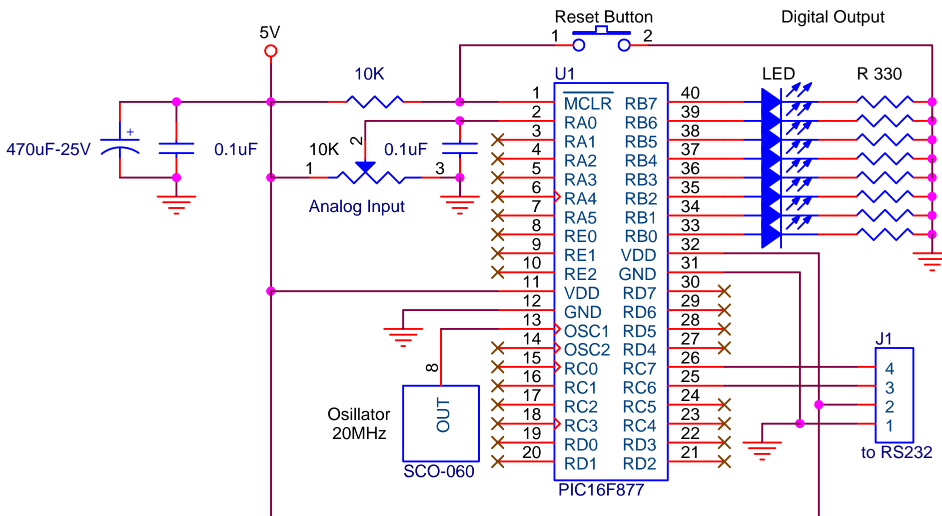
VD :

```
setup_adc(  ADC_CLOCK_INTERNAL  );
setup_adc_ports( ALL_ANALOG );
set_adc_channel(1);
while ( input(PIN_B0) )
{
  delay_ms( 5000 );
  value = read_adc();
  printf("A/D value = %2x\n\r", value);
}
read_adc(ADC_START_ONLY);
sleep();
value=read_adc(ADC_READ_ONLY);
```

_Lưu ý : trên PIC 18 , cấu trúc ADC tương đối phức tạp , đa năng hơn như là cho phép lấy 2 mẫu cùng lúc , . . . cũng sử dụng với các hàm trên , có nhiều thông số trong file *.h , sẽ để cập sau .

5 / _ Ví dụ :

_Chương trình sau lấy ADC 8 bit , đọc và xuất ra dãy led ở port B , và xuất ra màn hình máy tính .
_Kết nối chân trên 16F877 : RA0 là chân lấy Analog vào , áp chuẩn là nguồn +5V , mass=0 V
_Hình sau trích trong tài liệu thầy Nguyễn Tân Tiến viết T6-2002 .




```
#include <16F877.h >
#use delay( clock=20000000 )
#device *= 16  ADC = 8                    // sử dụng ADC 8 bit , giá trị ADC vào từ 0-255
#use rs232(baud=19200,parity=n,xmit=pin_C6,rcv=pin_C7)
Int8 adc ;
```

```
Main( )
{
Setup_ADC ( ADC_internal ) ;
Setup_ADC_ports (AN0);
Set_ADC_channel ( 0 ) ;
Delay_us (10 );                    // delay 10 us
While (true )
{       adc = read_adc ( ) ;
        Output_B ( adc ) ;          // xuat ra port B gia tri bien adc
        Printf( " gia tri adc la : %u " , adc ) ;      // in ra man hinh
}
}
```

// giá trị biến adc từ 0-255 , dùng chương trình Serial port Monitor trong mục Tools của CCS để giám sát giá trị . Nhớ thiết lập tốc độ là 19200 như khai báo trên .

II / _ CÁC HÀM VÀO RA TRONG C :

_Bao gồm các hàm sau :

Output_low()	Output_high()
Output_float()	Output_bit()
Input()	Ouput_X()
Input_X()	port_b_pullups()
Set_tris_X()	

1 / Output_low (pin) , Output_high (pin) :

_Dùng thiết lập mức 0 (low, 0V) hay mứ c 1 (high , 5V) cho chân IC , pin chỉ vị trí chân .
_Hàm này sẽ đặt pin làm ngõ ra , xem mã asm để biết cụ thể .
_Hàm này dài 2-4 chu kỳ máy . Cũng có thể xuất xung dùng set_tris_X() và #use fast_io.
VD : chương trình sau xuất xung vuông chu kỳ 500ms , duty =50% ra chân B0 ,nối B0 với 1 led sẽ làm nhấp nháy led .

```
#include <16F877.h>
#use delay( clock=20000000)
Main()
{       while(1)
        {       output_high(pin_B0) ;
                Delay_ms(250) ;             // delay 250ms
                Output_low (pin_B0);
                Delay_ms (250 );
        }
}
```

2 / Output_bit (pin , value) :

_pin : tên chân   value : giá trị 0 hay 1

_Hàm này cũng xuất giá trị 0 / 1 trên pin , tương tự 2 hàm trên . Thường dùng nó khi giá trị ra tuỳ thuộc giá trị biến 1 bit nào đó , hay muốn xuất đảo của giá trị ngõ ra trước đó .
VD :
Khai báo int1 x; // x mặc định = 0
Trong hàm main :

```
Main()
{       while (1 )
        {       output_bit( pin_B0 , !x ) ;
                Delay_ms(250 );
        }
}
```

Chương trình trên cũng xuất xung vuông chu kỳ 500ms ,duty =50%

3 / Output_float (pin) :

_Hàm này set pin như ngõ vào , cho phép pin ở mức cao như 1 cực thu hở (This will allow the pin to float high to represent a high on an open collector type of connection , dịch như vậy không biết đúng không nữa ? , chắc là thiết lập như ngõ vào tổng trở cao thì phải) .

4 / Input (pin) :

_Hàm này trả về giá trị 0 hay 1 là trạng thái của chân IC . Giá trị là 1 bit

5 / Output_X (value) :

_X là tên port có trên chip . Value là giá trị 1 byte .
_Hàm này xuất giá trị 1 byte ra port . Tất cả chân của port đó đếu là ngõ ra .
VD :
Output_B (212) ; // xuất giá trị 11010100 ra port B

6 / Input_X ( ) :

_X : là tên port (a, b ,c ,d e) .
_Hàm này trả về giá trị 8 bit là giá trị đang hiện hữu của port đó .VD : m=input_E();

7 / Port_B_pullups (value) :

_Hàm này thiết lập ngõ vào port B pullup (điện trở kéo lên ?) . Value =1 sẽ kích hoạt tính năng này và value =0 sẽ ngừng .
_Chỉ các chip có port B có tính năng này mới dùng hàm này .

8 / Set_tris_X (value) :

_Hàm này định nghĩa chân IO cho 1 port là ngõ vào hay ngõ ra. Chỉ được dùng với #use fast_IO . Sử dụng #byte để tạo biến chỉ đến port và thao tác trên biến này chính là thao tác trên port .
_Value là giá trị 8 bit . Mỗi bit đại diện 1 chân và bit=0 sẽ set chân đó là ngõ vào , bit= 1 set chân đó là ngõ ra .
VD : chương trình sau cho phép thao tác trên portB 1 cách dễ dàng:

```
#include < 16F877.h >
#use delay(clock=20000000)
#use Fast_IO( B )
#byte portB = 0x6          // 16F877 có port b ở địa chỉ 6h
#bit B0 = portB. 0         // biến B0 chỉ đến chân B0
#bit B1=portB.1            // biến B1 chỉ đến chân B1
```

```
#bit B2=portB.2              // biến B2 chỉ đến chân B2
#bit B3=portB.3              // biến B3 chỉ đến chân B3
#bit B4=portB.4              // biến B4 chỉ đến chân B4
#bit B5=portB.5              // biến B5 chỉ đến chân B5
#bit B6=portB.6              // biến B6 chỉ đến chân B6
#bit B7=portB.7              // biến B7 chỉ đến chân B7

Main()
{       set_tris_B ( 126 ) ;         //portB=01111110 b
                             // B0 là ngõ vào , thường làm ngắt ngoài
                             //B1 . . . B6 là ngõ ra , Vd làm 6 ngõ ra điều chế PWM
                             //B7 là ngõ vào , Vd là nhận tín hiệu cho phép chẳng hạn
        if ( B7 )            //nếu ngõ vào chân B7 là 1 thì xuất 3 cặp xung đối nghịch
        {       B1 = 1 ;
                B2 = 0 ;
                B3 = 1 ;
                B4 = 0 ;
                B5 = 1 ;
                B6 = 0 ;
        }
        Else  B1=B2=B3=B4=B5=B6= 0;
}
```

Lưu ý :

_Set_tris_B (0) : port B =00000000 : tất cả chân portB là ngõ ra
_set_tris_B (1) : portB = 00000001 : chỉ B0 là ngõ vào , còn lại là ngõ ra
_set_tris_B (255) : portB=11111111: tất cả chân portB là ngõ vào

- ➔ tôi cũng từng nhầm lẫn khi nghĩ set_tris_B(1) là set tất cả là ngõ vào , rất tai hại . Bạn nên dùng giá trị ở dạng nhị phân cho dễ . VD : set_tris_B (00110001b) ;

_Đến đây là bạn có thể viết nhiều chương trình thú vị rồi đó. Vd như là dùng ADC để điều chỉnh tốc độ nhấp nháy của dãy đèn led , truyền giá trị 8 bit từ chip này sang chip khác , . . .
_Chương trình VD sau dùng ADC qua chân A0 để điều chỉnh tốc độ nhấp nháy dãy đèn led nối vào port B , có thể dùng fast_io hay hàm output_B () để xuất giá trị đều được . chương trình dùng hàm . Nếu ngõ vào chân C0 =0 thì tiếp tục nhận ADC và xuất ra portB, C0=1 thì không xuất

```
#include <16F877.h>
#device *=16 ADC= 8
#use delay( clock =20000000)
Int8 ADC_delay  ;

Void hieu_chinh (  )
{       ADC_delay = read_adc ( 0 ) ;
        Output_B ( 0) ;              //portB=00000000
        Delay_ms ( ADC_delay );
        Output_B ( 255 ) ;           // portB= 11111111
        Delay_ms ( ADC_delay );
}
```

```
Main()
{
setup_adc_ports(AN0_AN1_AN3);          // A0 ,  A1 và  A3 là chân analog , ta chỉ cần dùng A0 lấy
                                          tín hiệu
setup_adc(adc_clock_internal);
set_adc_channel ( 0 );                  // chọn đọc ADC từ chân A0
while(1)
{       hieu_chinh ( ) ;
        If ( input ( pin_C0 )
        {       output_B (0 );
                Break ;                 // thoát khỏi vòng lặp while nhỏ
        }
}       //while
}       // main
```

CHƯƠNG 4 :

TRUYỀN THÔNG VỚI PC QUA CỔNG COM-RS232 - XỬ LÝ CHUỖI

_Chương này sẽ giúp bạn viết chương trình có sử dụng giao tiếp PC . Điều này rất cần thiết khi bạn muốn VĐK khi hoạt động có thể truyền dữ liệu cho PC xử lý , hoặc nhận giá trị từ PC để xử lý và điều khiển (dùng PC điều khiển động cơ , nhiệt độ , hay biến PC thành dụng cụ đo các đại lượng điện , Oscilocope , . . .) .
_Viết chương trình lập trình cho VĐK để giao tiếp máy tính là công việc rất phức tạp khi viết bằng ASM , rất khó hiểu đối với những người mới bắt đầu lập trình . Đặc biệt là khi viết cho những con VĐK không hỗ trợ từ phần cứng (8951 thì phải (?)) . Thật may là phần lớn PIC hiện nay đều hỗ trợ phần này nên việc lập trình có dễ dàng hơn . Nhưng nếu chương trình của bạn yêu cầu truyền hay nhận nhiều loại dữ liệu (số 8 , 16 ,32 bit , dương , âm , chuỗi , . . .) thì việc viết chương trình xử lý và phân loại chúng là điều “ kinh dị “ .

_Nhưng nếu lập trình ASM cho vấn đề này rồi thì bạn sẽ thấy sao dễ dàng quá vậy khi giải quyết vấn đề này với C khi dùng CCS . Rất đơn giản ! CCS cung cấp rất nhiều hàm phục vụ cho giao tiếp qua RS232 (cổng COM) và vô số hàm xử lý chuỗi . Chương này sẽ giải quyết điều đó .
_Một yếu tố quan trọng là khi nào thì VĐK biết PC truyền data → có thể lập trình bắt tay bằng phần mềm hay đơn giản là dùng ngắt . Các ví dụ về ngắt , xem phần ngắt .

I / _TRUYỀN THÔNG VỚI PC QUA CỔNG COM :

_Để sử dụng giao thức này , phải có 2 khai báo như VD sau :

```
#use delay (clock = 40000000 )                // VDK đang dùng OSC 40Mhz
#use rs232 (baud=19200 , parity=n , xmit=pin_C6 , rcv=pin_C7 )
        // baud= 19200 , không chẵn lẻ , chân truyền C6 , chân nhận C7
```

_Các hàm liên quan :

Printf ( )	
Getc ( )	putc ( )
Getch ( )	putchar ( )
Getchar ( )	fputc ( )
Fgetc ( )	puts ( )
Gets ( )	fputs ( )
Fgets ( )	
Kbhit ( )	

Assert ( ) →mới trên CCS 3.222
Perror ( ) → mới trên CCS 3.222
Set_uart_speed ( )
Setup_uart ( )

_Tất cả các hàm trên đòi hỏi phải khai báo chỉ thị tiền xử lý #use RS232 (. . . . .) .
_Hàm perror ( ) đòi hỏi thêm #include<errno.h > . Hàm assert() đòi hỏi thêm #include<assert.h> .

1 / printf (**string**)
Printf (**cstring** , **values** . . .)

_Dùng xuất chuỗi theo chuẩn RS232 ra PC .
_string là 1 chuỗi hằng hay 1 mảng ký tự (kết thúc bởi ký tự null) .
_value là danh sách các biến , cách nhau bởi dấu phẩy .
_Bạn phải khai báo dạng format của value theo kiểu %wt .Trong đó w có thể có hoặc không , có giá trị từ 1-9 chỉ rõ có bao nhiêu ký tự được xuất ra (mặc định không có thì có bao nhiêu ra bấy nhiêu), hoặc 01-09 sẽ chèn thêm 0 cho đủ ký tự hoặc 1.1-1.9 cho trường hợp số thực . còn t là kiểu giá trị .
_ t có thể là :
C : 1 ký tự
S : chuỗi hoặc ký tự
U : số 8 bit không dấu
x : số 8 bit kiểu hex (ký tự viết thường ,VD : 1ef)
X : số 8 bit kiểu hex (ký tự viết hoa ,VD : 1EF)
D : số 8 bit có dấu
e : số thực có luỹ thừa VD : e12
f : số thực

Lx : số hex 16 /32 bit (ký tự viết thường)
LX : hex 16 /32 bit (ký tự viết hoa)
Lu : số thập phân không dấu
Ld : số thập phân có dấu
% : ký hiệu %
VD :

Specifier	Value=0x12	Value=0xfe
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2
%x	12	Fe
%X	12	FE
%4X	0012	00FE

* Result is undefined - Assume garbage.

VD :
Int k =6 ;
Printf (" hello ");
Printf (" %u " , k);

2 / KBHIT ( ) :

_Thường thì chúng ta dùng RC6 và RC7 cho RX và TX trong giao tiếp cổng COM , VDK PIC trang bị phần cứng phục vụ việc này với thanh ghi gởi và nhận và các bit bào hiệu tương ứng . Do đó khi dùng RS232 hỗ trợ từ phần cứng thì KHBIT ( ) trả về TRUE nếu 1 ký tự đã được nhận (trong bộ đệm phần cứng) và sẵn sàng cho việc đọc , và trả về 0 nếu chưa sẵn sàng .
_Hàm này có thể dùng hỏi vòng xem khi nào có data nhận từ RS232 để đọc .

CHƯƠNG 6 :

GIAO TIẾP SPI – I2C VÀ PARALLEL

I / GIAO TIẾP SPI :

_Đây là giao tiếp dễ dùng nhất , đơn giản nhất , tốc độ cao nhất trong nhóm . hoạt động theo cơ chế hand-shaking , bắt tay . Giả sử có 2 VDK , thì 1 là master , 1 là slave . Khi master truyền 1 byte cho slave , nó phát 8 xung clock qua đường clock nối tới slave , đồng thời truyền 8 bit data từ chân SDO

tới chân SDI của slave . Không kiểm tra chẵn lẻ , lỗi . Do đó Vdụ nếu đang truyền được 3 bit mà master reset hay hở dây clock thì data bị mất , slave sẽ không nhận đủ 8 bit và do đó nếu tiếp tục nhận nó sẽ lấy 5 bit ở byte kế tiếp đưa vào thanh ghi nhận để đủ 8 bit (và để kích ngắt) . Từ đó trở đi là mọi giá trị nhận là sai bét trừ phi chấm dứt và sau đó thiết lập lại giao tiếp này (ở cả hai) .
_Giao tiếp này cần ít nhất 2 dây trở lên . Nếu 1 VDK chỉ cần gởi data thì chỉ cần dây clock và SDO .VDK nhận sẽ dùng SDI và dây clock . Dây clock là nối chung .
_Nếu có gởi và nhận ở cả 2 VDK thì : dây clock chung , master có SDO nối tới SDI của slave , SDO của slave nối tới SDI của master .
_Nếu master cần truyền data cho nhiều slave trở lên thì SDO master nối tới các SDI của slave .
_Chân SS là slave select .
_SPI hoạt động từ phần cứng , vì nó có sẵn thanh ghi gởi và nhận , nhận đủ giá trị thì có cờ ngắt phục vụ .
_Danh sách các hàm :

1 / Setup_spi (mode)
Setup_spi2 (mode)

_Dùng thiết lập giao tiếp SPI . Hàm thứ 2 dùng với VDK có 2 bộ SPI .
_Tham số mode :là các hằng số sau , có thể OR giữa các nhómbởi dấu |

→ SPI_MASTER , SPI_SLAVE , SPI_SS_DISABLED
→ SPI_L_TO_H , SPI_H_TO_L
→SPI_CLK_DIV_4 , SPI_CLK_DIV_16 , SPI_CLK_DIV_64 , SPI_CLK_T2

_Nhóm 1 xác định VDK là master hay slave ,slave select
_Nhóm 2 xác định clock cạnh lên hay xuống .
_Nhóm 3 xác định tần số xung clock , SPI_CLK_DIV_4 ngĩa là tần số = FOSC / 4 , tương ứng 1 chu kỳ lệnh / xung .
_Hàm không trả về trị .
_Ngoài ra ,tuỳ VDK mà có thêm 1 số tham số khác , xem file * .h .

2 / Spi_read (data)
Spi_read2 (data)

_data có thể có thêm và là số 8 bit . Hàm thứ 2 cho bộ SPI thứ 2 .
_Hàm trả về giá trị 8 bit value = spi_read ( )
_Hàm trả về giá trị đọc bởi SPI . Nếu value phù hợp SPI_read ( ) thì data sẽ được phát xung ngoài và data nhận được sẽ được trả về . Nếu không có data sẵn sàng , spi_read ( ) sẽ đợi data .
_Hàm chỉ dùng cho SPI hardware (SPI phần cứng) .

3 / spi_write (value)
Spi_write2 (value)

_Hàm không trả về trị . value là giá trị 8 bit .
_Hàm này gửi value (1 byte) tới SPI , đồng thời tạo 8 xung clock .
_ Hàm chỉ dùng cho SPI hardware (SPI phần cứng) .

4 / spi_data_is_in ( )
Spi_data_is_in2 ( )

_Hàm trả về TRUE (1) nếu data nhận được đầy đủ (8 bit) từ SPI , trả về false nếu chưa nhận đủ .
_Hàm này dùng kiểm tra xem giá trị nhận về SPI đã đủ 1 byte chưa để dùng hàm spi_read ( ) đọc data vào biến .

CHƯƠNG 7 :

MODULE PWM / CAPTURE / COMPARE

I / _TỔNG QUÁT PHẦN CỨNG :

_Module này có mặt ở hầu hết các dòng PIC 16 và PIC 18 , và thường chỉ có 2 chân cho module này , ký hiệu là CCP1 / CCP2 . Cần phân biệt với module PWM chuyên dụng trên PIC 18 và dsPIC vốn có 6 tới 8 chân PWMx , cũng phục vụ cho điều chế độ rộng xung nhưng chuyên dụng cho điều khiển động cơ AC . Một số PIC 14 và PIC 18 có module ECCP cũng dùng module CCP này nhưng lại chuyên dụng cho điều khiển bộ biến đổi áp DC->DC , dùng cho điều khiển động cơ DC . Ở đây chưa đề cập đến ECCP . Chỉ đề cập CCP , và thường chỉ có 2 chân là CCP1 và CCP2 .
_Module có 3 chức năng và khi hoạt động ta chỉ dùng được 1 chức năng ứng với 1 chân . Ngoài ra nếu không dùng thì có thể set nó thành chân I /O .
_Mỗi module chứa 1 thanh ghi 16 bit , là kết hợp của 2 thanh ghi 8 bit : CCPR1L(byte thấp) và CCPR1H (byte cao) của CCP1 , CCPR2L và CCPR2H của CCP2 .
_Mỗi chức năng của CCPx đều đòi hỏi 1 bộ đếm để hoạt động : Capture / Compare đòi hỏi Timer1 , còn PWM đòi hỏi Timer2 .

TABLE 8-2: INTERACTION OF TWO CCP MODULES

CCPx Mode	CCPy Mode	Interaction
Capture	Capture	Same TMR1 time-base
Capture	Compare	The compare should be configured for the special event trigger, which clears TMR1
Compare	Compare	The compare(s) should be configured for the special event trigger, which clears TMR1
PWM	PWM	The PWMs will have the same frequency and update rate (TMR2 interrupt)
PWM	Capture	None
PWM	Compare	None

Bảng trên (trong datasheet PIC16F877) cho thấy : nếu cả 2 module dùng cùng chức năng Capture thì sẽ dùng chung bộ đếm timer1 , nghĩa là mọi hoạt động và điều chỉnh Timer1 đều ảnh hưởng tới cả 2 module . Tương tự nếu dùng cùng chức năng PWM sẽ dùng chung Timer2 . Còn nếu 1 module là Capture , module kia là Compare thì phải lưu ý là chức năng compare có thể clear Timer1 , và do đó cũng ảnh hưởng chức năng còn lại . Còn 2 TH cuối thì không có vấn đề gì .
_Ở chế độ Capture : mỗi khi có cạnh lên của xung vào chân CCPx thì giá trị Timer1 (16 bit)sẽ được copy vào thanh ghi CCP (16 bit) .
_Ở chế độ Compare : khi giá trị nạp CCP bằng giá trị đếm Timer1 thì các sư kiện được chỉ định trước xảy ra :chân CCPx được lái ra mức thấp / cao / không có gì nhưng có ngắt hay biến đổi AD .
_Chế độ PWM (pulse width modulation – điều chế độ rộng xung) : xuất xung vuông bằng phần cứng .
_Bạn sử dụng hàm setup_ccpX() để xác định chức năng muốn dùng trên module CCP . tham số cụ thể có thể tra trong HELP , hoặc xem từng TH cụ thể dưới đây .

II / _CHẾ ĐỘ CAPTURE :

_Dùng để xác định tốc độ quay của motor . Giả sử ta có 1 thiết bị đo tốc độ quay (cảm biến Hall) mà mỗi khi motor quay 1 vòng thì thiết bị phát 1 xung vuông (cạnh lên rồi cạnh xuống hay xung mức 1 , hay cạnh xuống rồi cạnh lên tức là xung mức 0) . Ở đây ta giả thiết xung mức 1 . Xung này dẫn vào module capture (có thể qua cách ly an toàn) và giả thiết ta chọn chế độ bắt cạnh lên . Nghĩa là cứ mỗi 1 cạnh lên của xung vào , giá trị của timer1 copy vào CCP . Vì motor không phải quay đều nên ta thường chọn số lần capture là vài chục lần . (hình như là vậy ???)

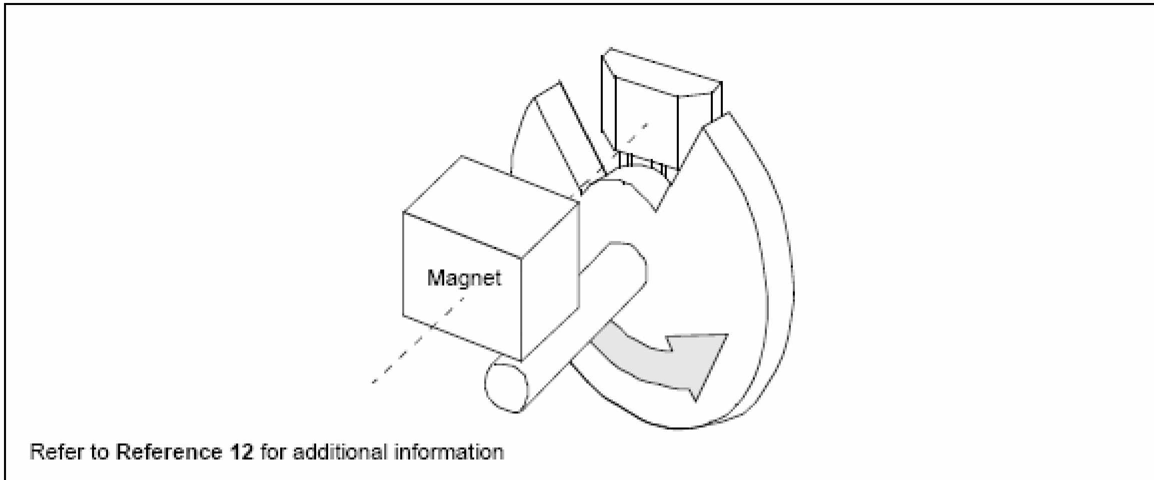


FIGURE 18: Hall Effect Rotary Interrupt Switch Tachometer.

_Việc Capture thường đi kèm 1 cái hàm ngắt . Cách thức chương trình hoạt động như sau : xung vào Capture sẽ kích ngắt , trong hàm ngắt ,ta lấy giá trị CCPx tính ra được thời gian cho 1 vòng quay ,cộng dồn nó vào 1 biến để tính thời gian tổng , set Timer1 về 0 , tăng giá trị biến đếm vòng lên 1 , rồi thoát ngắt . Với 1 động cơ đang chạy khoảng 1200vòng / phút , tuỳ thuộc vào tần suất cập nhật giá trị số vòng quay (để hiển thị ra LED 7 đoạn hay LCD) bạn cần đo 1 số lượng vòng nhất định để đảm bảo chính xác , VD đo 1500 vòng , tức là biến đếm phải đếm tới 1500 , tính tổng thời gian đó , VD là 80 s , thì tốc độ động cơ = (1500 / 80) * 60 = 1125 vòng / phút .
_Hãy quên module này đi nếu bạn dùng 1 encoder để đo tốc , vốn phát ra tới 500-2000 xung / vòng .Nghĩa là cần chừng đó lần ngắt để chỉ đo 1 vòng → VDK sẽ phải dùng toàn bộ thời gian hoạt động để đếm , có khi không kịp . Việc chương trình chính không thể chạy vì ngắt cứ xảy ra liên tục gọi là tràn , nó sẽ chẳng làm được gì cả . Để giải quyết , người ta dùng module QEI hay IC3 vốn chỉ có trên PIC 18 như 18F4431 .
_Capture còn dùng để đo độ rộng xung . Sử dụng CCP1 lấy cạnh lên , CCP2 lấy cạnh xuống của cùng 1 xung ,tức là 2 CCP nối nhau . 1 xung vuông đi vào kích CCP1 trước (cạnh lên), thu được giá trị timer lúc , cạnh xuống của xung kích CCP2 thu được giá trị timer lúc xuống cạnh . Lấy 2 giá trị trừ nhau được độ rộng xung . chỉ cần 1 ngắt CCP2 là đủ .

III / _CHẾ ĐỘ COMPARE :

_Hoạt động : khi timer1 đếm lên tới khi bằng giá trị mà ta trữ trong CCPx, thì chân CCPx sẽ xuất ra mức cao / thấp / hay không có nhưng có ngắt .
_Ứng dụng : không rõ . Bạn nào biết bổ sung .

IV / _CHẾ ĐỘ PWM :

_Xuất chuỗi xung vuông , độ rộng hiệu chỉnh được dễ dàng . Thường dùng để điều chỉnh điện áp DC . Xung ra sẽ đóng cắt 1 linh kiện như là SCR, với độ rộng xung xác định sẽ tạo ra 1 điện áp trung bình xác định . Thay đổi độ rộng xung sẽ thay đổi điện áp này , do đó có thể dùng điều khiển động cơ DC nhỏ (? ? ?) .

V / _DANH SÁCH HÀM:

_ CCS luôn tạo sẵn các tên danh định C như là các biến trỏ tới CCP1 và CCP2 là : CCP_1 (16 bit) , CCP_2 (16 bit) , CCP_1_HIGH (byte cao của CCP1) , CCP_1_LOW , CCP_2_HIGH ,

CCP_2_LOW , bạn không cần khai báo . Dùng luôn các tên đó để lấy trị khi dùng module Cap , hay gán trị khi dùng Compare . Bạn có thể thấy điều này khi mở mục RAM symbol map quan sát phân bổ bộ nhớ .

1 / _ Setup_CCPx (mode) :

_Dùng trước tiên để thiết lập chế độ hoạt động hay vô hiệu tính năng CCP .
X= 1,2, . . .tên chân CCP có trên chip .
Mode là 1 trong các hằng số sau : (các hằng số khác có thể có thêm trong file *. h và tuỳ VDK)

CCP_OFF : tắt chức năng CCP , RC sẽ là chân I /O .
CCP_CAPTURE_RE : capture khi có cạnh lên
CCP_CAPTURE_FE : capture khi có cạnh xuống
CCP_CAPTURE_DIV_4 : chỉ capture sau khi đếm đủ 4 cạnh lên (4 xung).
CCP_CAPTURE_DIV_16 : chỉ capture sau khi đếm đủ 16 cạnh lên (16 xung).

→ sử dụng để làm dãn thời gian VDK để dành cho công việc khác thay vì cứ update từng xung .
Chế độ compare :

CCP_COMPARE_SET_ON_MATCH : xuất xung mức cao khi TMR1=CCPx
CCP_COMPARE_CLR_ON_MATCH : xuất xung mức thấp khi TMR1=CCPx
CCP_COMPARE_INT : ngắt khi TMR1=CCPx
CCP_COMPARE_RESET_TIMER : reset TMR1 =0 khi TMR1=CCPx

Chế độ PWM :

CCP_PWM : bật chế độ PWM
CCP_PWM_PLUS_1 : không rõ chức năng
CCP_PWM_PLUS_2 : không rõ chức năng
CCP_PWM_PLUS_3 : không rõ chức năng

2 / _ Set_CCPx_duty (value) :

Value : biến hay hằng , giá trị 8 hay 16 bit .
x= 0 ,1 ,2 . . . :tên chân CCPx
_Dùng set duty của xung trong chế độ PWM . Nó ghi 10 bit giá trị vào thanh ghi CCPx . Nếu value chỉ là 8 bit, nó dịch thêm 2 để đủ 10 bit nạp vào CCPx .
_Tuỳ độ phân giải mà giá trị của value không phải lúc nào cũng đạt tới 1023 . Do đó , value = 512 không có nghĩa là duty = 50 % .

CHƯƠNG 8 :

LÀM VIỆC VỚI NGẮT

I / CƠ CHẾ HOẠT ĐỘNG CỦA NGẮT :

1 / _ Ngắt 1 cấp :

_Trên PIC 14 , 12 ,10 ,tất cả các ngắt chỉ có 1 cấp ưu tiên . Nghĩa là ngắt nào đang được phục vụ thì không thể bị ngắt bởi 1 ngắt khác xảy ra . Cơ chế sinh mã cho ngắt của CCS như sau : nhảy đến địa chỉ ngắt , thường là 004h , sao lưu thanh ghi W, STATUS , PCLATCH , FSR, và nhiều thứ vớ vẩn khác, sau đó nó mới hỏi vòng xem cờ ngắt nào xảy ra thì nhảy đến hàm phục vụ ngắt đó . thực hiện xong thì phục hồi tất cả thanh ghi trên , rồi mới "RETFIE" – thoát ngắt . Số chu kỳ thực thi từ chỗ ngắt đến khi nhảy vào hàm ngắt cỡ 20 chu kỳ lệnh !, nhảy ra cũng cỡ đó .

_Điều gì xảy ra nếu chương trình dùng nhiều ngắt và khi có ngắt thì có 2 ngắt trở lên xảy ra đồng thời ? Nghĩa là : 2 ngắt xảy ra cùng lúc , hay khi ngắt A kích hoạt và CCS đang lưu các thanh ghi (chưa tới hỏi vòng cờ ngắt) thì ngắt B xảy ra , dĩ nhiên ngắt B không thể kích vector ngắt nhảy tới 004h vì bit cho phép ngắt toàn cục (GIE) bị khóa tự động khi có ngắt , chỉ có cờ ngắt B bật mà thôi. Sau khi lưu các thanh ghi , chương trình kiểm tra cờ ngắt , rõ ràng là nếu bit nào được kiểm tra trước thì phục vụ trước , dù nó xảy ra sau . Để tránh phục vụ không đúng chỗ , bạn dùng #priority để xác định ưu tiên ngắt (xem phần chỉ thị tiền xử lý) . Ngắt ưu tiên nhất sẽ luôn được hỏi vòng trước .Sau khi xác định cờ ngắt cần phục vụ , nó sẽ thực thi hàm ngắt tương ứng .Xong thì xoá cờ ngắt đó và thoát ngắt . Phục vụ ngắt nào xong thì chỉ xoá cờ ngắt đó .Nếu A ưu tiên hơn B thì sau khi làm A , chương trình xoá cờ ngắt A , nhưng cờ B không xoá (vì đâu có phục vụ) , nên khi thoát ra ngắt A , nó sẽ lại ngắt tiếp (vì cờ B đã bật), lại hỏi vòng cờ ngắt từ đầu : nếu cờ A chưa bật thì xét B, lúc này B bật nên phục vụ B , xong thì xoá cờ B và thoát ngắt .

_Môt chương trình dùng nhiều ngắt phải lưu ý điều này , tránh trường hợp : ngắt xảy ra liên tục (tràn ngắt) , 1 ngắt bị đáp ứng trễ , ngắt không đúng , . . .

2 / _ Ngắt 2 cấp :

_Chỉ có trên PIC 18 (và dsPIC) . Có 2 khái niệm : ngắt ưu tiên thấp (low priority) và ngắt ưu tiên cao (high priority) . 2 vector thực thi ngắt tương ứng thường là 0008h (high) và 0018h (low) . Một ngắt thấp đang được phục vụ sẽ bị ngưng và phục vụ ngắt cao ở 0008h nếu ngắt cao xảy ra . Ngược lại , ngắt cao đang xảy ra thì không bao giờ bị ngắt bởi ngắt thấp .

_Nếu viết hàm ngắt bình thường , không đòi hỏi ưu tiên gì thì CCS sinh mã để tất cả hàm ngắt đều là ngắt ưu tiên cao . Quy trình thực hiện ngắt sẽ như ngắt 1 cấp trên . #priority vẫn được dùng . Số chu kỳ thực thi từ 0008h đến khi nhảy vào thực thi hàm ngắt khoảng 30 chu kỳ , xong hàm ngắt tới khi kết thúc ngắt cũng mất khoảng 30 chu kỳ lệnh .

_Để sử dụng ngắt 2 cấp , khai báo #device phải có high_ints=true . Và hàm ngắt nào muốn ưu tiên cao thì thêm FAST theo sau chỉ thị tiền xử lý hàm đó . Lưu ý : chỉ có duy nhất 1 ngắt được ưu tiên cao , đây có lẽ là hạn chế của CCS , do cách thức sinh mã .

VD :
```
#int_timer1 FAST
Void xu_ly ( )
{ . . .
}
```

_Cơ chế sinh mã như sau : có ngắt thấp thì nhảy tới 0018h , sao lưu W, STATUS , FSR0/1/2 ,. . . rồi mới hỏi vòng cờ ngắt thấp . chạy xong hàm ngắt thì phục hồi tất cả và "RETFIE 0 " . Riêng ngắt

cao không sinh mã sao lưu gì cả mà nhảy thẳng vào hàm ngắt chạy luôn . Vậy thì trật lất rồi ? Mã chạy sai chăng ?

_Thực ra không phải vậy . PIC 18 và dsPIC có cơ chế lưu siêu tốc là FAST STACK REGISTER (xem datasheet kỹ nhé) . Khi xảy ra ngắt bất kỳ , W, S , BSR tự động lưu vào thanh ghi trên , PC counter lưu vào stack . xong ngắt thì pop ra . Vấn đề ở chỗ : khi ngắt thấp xảy ra , FAST STACK REGISTER tự động lưu W ,S , BSR , PC -> stack . Trong khi thực hiện hàm phục vụ ngắt thì trường hợp W, S , BSR thay đổi là có thể (vì vậy mới sao lưu chứ) . nhưng nếu xảy ra ngắt cao vào thời điểm đó ? FAST STACK REGISTER sẽ bị ghi đè → mất data . Do đó , cơ chế sinh mã của CCS cần phải luôn đúng , nghĩa là : luôn tự sao lưu riêng W ,S , BSR, và các thanh ghi FSR nữa , khi thực thi ngắt thấp . Còn ngắt cao khi chạy xong sẽ "RETFIE 1 " – tự động phục hồi W, S , BSR từ FAST STACK REGISTER . Có 2 trường hợp : 1 là chỉ có ngắt cao , thì không có vấn đề gì . 2 là ngắt cao ngắt 1 ngắt thấp đang chạy . Phân tích sẽ thấy rằng cho dù bị ngắt trong khi đang sao lưu ,hay chưa kịp sao lưu , hay đã sao lưu vào các biến riêng rồi , cuối cùng chương trình cũng quay ra đúng địa chỉ ban đầu với các thanh ghi W, S , BSR như cũ .

_Tuân thủ nguyên tắc ngắt cao thực thi tức thời nên CCS chỉ cho 1 ngắt cao duy nhất bất kỳ hoạt động , nên không sinh mã hỏi vòng , sao lưu thêm gì cả . nếu bạn muốn có nhiều ngắt ưu tiên cao , thì phải tự viết mã riêng thôi (khi có ngắt cao thì hỏi vòng các cờ ngắt , dùng lệnh ORG chiếm đoạn mã từ 0008h trở đi để viết mã xử lý riêng , trong chương trình không được viết bất kỳ hàm ngắt nào kể cả ngắt thấp mà chỉ viết hàm bình thường , . . . nói chung là tự xử lý hết mọi vấn đề ngắt , phức tạp lắm đấy) .

II / KHAI BÁO NGẮT :

_Mỗi dòng VDK có số lượng ngắt khác nhau : PIC 14 có 14 ngắt , PIC 18 có 35 ngắt .

_Muốn biết CCS hỗ trợ những ngắt nào cho VDK của bạn , mở file *.h tương ứng , ở cuối file là danh sách các ngắt mà CCS hỗ trợ nó . Cách khác là vào CCS -> View -> Valid interrupts , chọn VDK muốn xem , nó sẽ hiển thị danh sách ngắt có thể có cho VDK đó .

_Sau đây là danh sách 1 số ngắt với chức năng tương ứng :

#INT_GLOBAL : ngắt chung , nghĩa là khi có ngắt xảy ra , hàm theo sau chỉ thị này được thực thi , bạn sẽ không được khai báo thêm chỉ thị ngắt nào khác khi sử dụng chỉ thị này . CCS không sinh bất kỳ mã lưu nào , hàm ngắt bắt đầu ngay tại vector ngắt . Nếu bật nhiều cờ cho phép ngắt , có thể bạn sẽ phải hỏi vòng để xác định ngắt nào . Dùng chỉ thị này tương đương viết hàm ngắt 1 cách thủ công mà thôi , như là viết hàm ngắt với ASM vậy .

#INT_AD : chuyển đổi A /D đã hoàn tất , thường thì không nên dùng
#INT_ADOF : I don't know
#INT_BUSCOL : xung đột bus
#INT_BUTTON : nút nhấn (không biết hoạt động thế nào)
#INT_CCP1 : có Capture hay compare trên CCP1
#INT_CCP2 : có Capture hay compare trên CCP2
#INT_COMP : kiểm tra bằng nhau trên Comparator
#INT_EEPROM : hoàn thành ghi EEPROM
#INT_EXT : ngắt ngoài
#INT_EXT1 : ngắt ngoài 1
#INT_EXT2 : ngắt ngoài 2
#INT_I2C : có hoạt động I 2C
#INT_LCD : có hoạt động LCD
#INT_LOWVOLT : phát hiện áp thấp
#INT_PSP : có data vào cổng Parallel slave

#INT_RB : bất kỳ thay đổi nào trên chân B4 đến B7
#INT_RC : bất kỳ thay đổi nào trên chân C4 đến C7
#INT_RDA : data nhận từ RS 232 sẵn sàng
#INT_RTCC : tràn Timer 0
#INT_SSP : có hoạt động SPI hay I 2C
#INT_TBE : bộ đệm chuyển RS 232 trống
#INT_TIMER0 : một tên khác của #INT_RTCC
#INT_TIMER1 : tràn Timer 1
#INT_TIMER2 : tràn Timer 2
#INT_TIMER3 : tràn Timer 3
#INT_TIMER5 : tràn Timer 5
#INT_OSCF : lỗi OSC
#INT_PWMTB : ngắt cuả PWM time base
#INT_IC3DR : ngắt đổi hướng (direct) của IC 3
#INT_IC2QEI : ngắt của QEI
#INT_IC1 : ngắt IC 1
_Hàm đi kèm phục vụ ngắt không cần tham số vì không có tác dụng .
_Sử dụng NOCLEAR sau #int_xxx để CCS không xoá cờ ngắt của hàm đó .
_Để cho phép ngắt đó hoạt động phải dùng lệnh enable_interrupts (int_xxxx) và enable_interrupts (global) .
_Khoá FAST theo sau #int_xxxx để cho ngắt đó là ưu tiên cao , chỉ được 1 ngắt thôi , chỉ có ở PIC 18 và dsPIC .
VD : #int_timer0 FAST NOCLEAR

III / CÁC HÀM THIẾT LẬP HOẠT ĐỘNG NGẮT :

1 / enable_interrupts (level)

_level là tên các ngắt đã cho ở trên hay là GLOBAL để cho phép ngắt ở cấp toàn cục .
_Mọi ngắt của VDK đều có 1 bit cờ ngắt , 1 bit cho phép ngắt . Khi có ngắt thì bit cờ ngắt bị set =1, nhưng ngắt có hoạt động được hay không tuỳ thuộc bit cho phép ngắt . enable_interrupts (int_xxx) sẽ bật bit cho phép ngắt . Nhưng tất cả các ngắt đều không thể thực thi nếu bit cho phép ngắt toàn cục = 0 , enable_interrupts(global) sẽ bật bit này .
VD : để cho phép ngắt timer0 và timer1 hoạt động:

```
enable_interrupts (int_timer0);
enable_interrupts (int_timer1 ) ;
enable_interrupts ( global );          // chỉ cần dùng 1 lần trừ phi muốn có thay đổi đặc biệt
```

2 / disable_interrupts (level)

_level giống như trên .
_Hàm này vô hiệu 1 ngắt bằng cách set bit cho phép ngắt = 0 .
_disable_interrupts (global) set bit cho phép ngắt toàn cục =0 , cấm tất cả các ngắt .
_Không dùng hàm này trong hàm phục vụ ngắt vì không có tác dụng , cờ ngắt luôn bị xoá tự động .

3 / clear_interupt (level)

_level không có GLOBAL .
_Hàm này xoá cờ ngắt của ngắt được chỉ định bởi level .

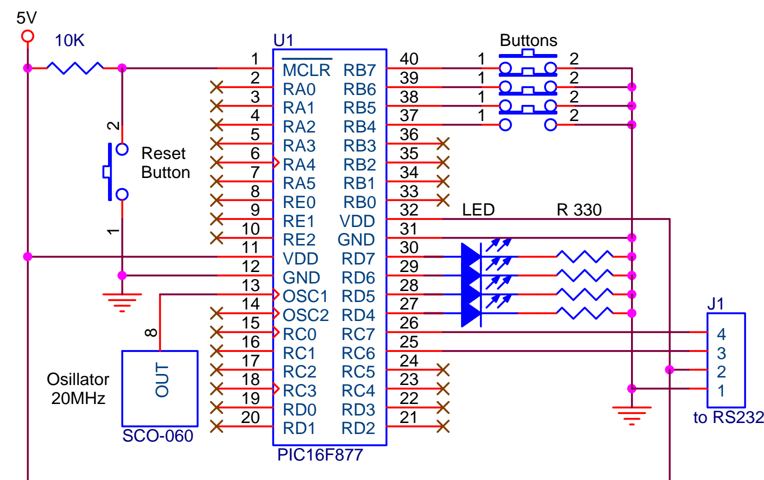
4 / ext_int_edge (source , edge)

_Hàm này thiết lập nguồn ngắt ngoài EXTx là cạnh lên hay cạnh xuống .
_source : nguồn ngắt . Trên PIC 18 có 3 nguồn ngắt trên 3 chân EXT0 , EXT1 , EXT2 ứng với source = 0 ,1 , 2 . Các PIC khác chỉ có 1 nguồn EXT nên source = 0 .
_edge : chọn cạnh kích ngắt , edge = L_TO_H nếu chọn cạnh lên (từ mức thấp chuyển lên mức cao) hay H_TO_L nếu chọn cạnh xuống .

IV / CÁC CHƯƠNG TRÌNH VD VỀ NGẮT :

1 / _ #INT_RB :

_Sau đây là 1 chương trình điển hình về sử dụng ngắt khi có sự thay đổi trên chân B4-B7 .
_Mô tả : mỗi khi nhấn nút bất kỳ trên B4-B7 , sẽ kích ngắt RB , hàm phục vụ ngắt có tên RB_LED được thực thi , hàm này đơn giản là xuất ra LED ở vị trí tương ứng nhưng trên portD từ D4 – D7 .
_VDK là 16F877 .




```
#include < 16F877.h >
#device  PIC16F877 *=16
#use delay (clock = 20000000 )        //thêm khai báo này nếu ctrình có dùng hàm delay,OSC=20 Mhz
#byte portb = 0x06                    //tạo tên danh định portb thay thế địa chỉ portB là 06h
#byte portd = 0x08                    //tạo tên danh định portd thay thế địa chỉ portD là 08h

#INT_RB
Void RB_LED ( )                       // hàm phục vụ ngắt
{
  portd=portb;
}

void main ( )
{       set_tris_b ( 0xF0 ) ;         // portB = 11110000 , B4-B7 là ngõ vào , B0-B3 là ngõ ra
        set_tris_d ( 0x00 ) ;         // portD = 00000000 , D0-D7 đều là ngõ ra
        enable_interrupts ( INT_RB ) ;        // cho phép ngắt RB
        enable_interrupts ( GLOBAL ) ;        // cho phép ngắt toàn cục
// do chương trình không làm gì khác ngoài việc chờ ngắt nên vòng while này trống không
```

```
while( true )
  { //có thể thêm mã xử lý ở đây . . .
  }
} //main
```