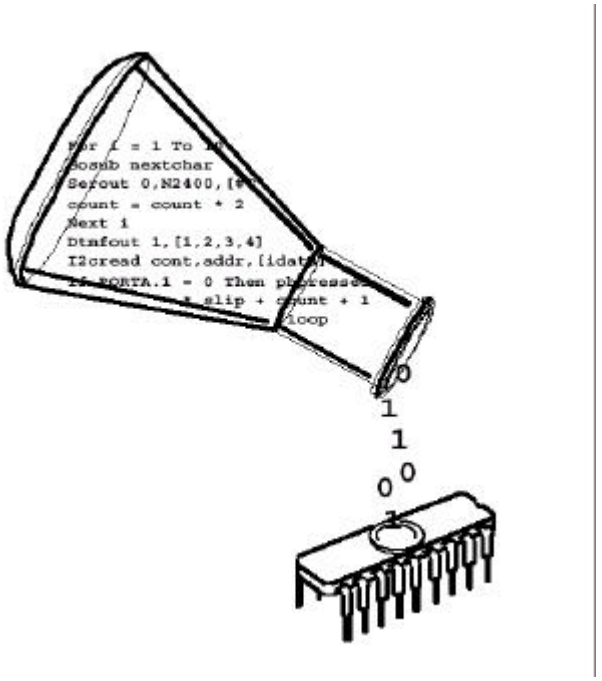
MANUAL DE USUARIO DEL COMPILADOR PCW DE CCS â



C Compiler for Microchip PICmicro® MCUs

Escrito por Andrés Cánovas López
Reeditado para formato PDF por Víctor Dorado

ÍNDICE

1. INTRODUCCIÓN

Si queremos realizar la programación de los microcontroladores PIC en un lenguaje como el C, es preciso utilizar un *compilador de C*.

Dicho compilador nos genera ficheros en formato Intel-hexadedimal, que es el necesario para programar (utilizando un programador de PIC) un microcontrolador de 6, 8, 18 ó 40 patillas.

El compilador de C que vamos a utilizar es el PCW de la casa CCS Inc. A su vez, el compilador lo integraremos en un e*ntorno de desarrollo integrado* (IDE) que nos va a permitir desarrollar todas y cada una de las fases que se compone un proyecto, desde la edición hasta la compilación pasando por la depuración de errores. La última fase, a excepción de la depuración y retoques hardware finales, será programar el PIC.

Al igual que el compilador de Turbo C, éste "*traduce*" el código C del archivo fuente (.C) a lenguaje máquina para los microcontroladores PIC, generando así un archivo en formato hexadecimal (.HEX). Además de éste, también genera otros seis ficheros, tal como se observa en la figura de la siguiente página.

Finalmente, deciros que esta vez os presento los apuntes en soporte

electrónico, a diferencia de ocasiones anteriores que estaban en formato impreso. Es una experiencia nueva y que como toda prueba tiene sus riesgos, aunque espero y deseo que, de una u otra forma logre prender la llama de la ilusión, o por lo menos despertar el interés por el estudio de la electrónica y en particular de este mundo inacabado de la programación en C y los microcontroladores.

2. PROGRAMAS DE UTILIDAD

- **SIO**

  SIO (Serial Input Output) es un simple programa "terminal no inteligente" que puede ejecutarse desde el DOS para realizar entradas y salidas sobre un puerto serie. SIO es útil ya que muestra todos los caracteres entrantes, excepto los no imprimibles que mostrará su código hexadecimal en rojo.

- **PICCHIPS**

  PICCHIPS es un programa de utilidad que lee la base de datos de un dispositivo. El compilador utiliza esta base de datos para determinar las características específicas del dispositivo durante la compilación. Al ejecutar el programa sin ningún parámetro, listará todos los dispositivos (PIC) disponibles. Si especificamos un dispositivo como parámetro p.ej. pic16c84, es decir, escribimos picchips pic16c84, obtenemos información detallada sobre este dispositivo. A modo de ejemplo y para el citado PIC se obtiene la siguiente información:

  PIC16C84---------- Opcode:

  14 bits, ROM: 1024, RAM: 36, I/O: 13 H/W: EEPROM(64) POR TIM0 TRIS
  RAM: 0C-2F
  Ports: [A:01234---] [B:01234567] [C:--------] [D:--------] [E:--------]
  Fuses: LP: 0003/0000 XT: 0003/0001
  HS: 0003/0002 RC: 0003/0003
  NOWDT: 0004/0000 WDT: 0004/0004
  NOPUT: 0008/0000 PUT: 0008/0008
  PROTECT: 3FF0/0000 NOPROTECT: 3FF0/3FF0
  ID is at 2000
  Par Device value: 0084
  C Device value: 84, C-Scratch at: 0C

- **CHIPEDIT**

  ChipEdit es una utilidad de Windows (sólo para PCW) que permite editar la base de datos de un dispositivo. Con esta utilidad podemos agregar dispositivos, modificarlos o eliminarlos de la base de datos. Para agregar un dispositivo, seleccionar de la lista otro equivalente, de caracteríticas similares, y pulsar el botón ADD. Para editar o borrar un dispositivo, seleccionarlo y pulsar el botón EDIT o DELETE.

- **CONVERT**

  PConvert es una utilidad de Windows (PCW sólo) que permite realizar conversiones de un tipo de datos a otros tipos. Por ejemplo, de decimal en Punto Flotante a Hexadecimal de 4 byte. La utilidad abre una ventana peque¤a para realizar las conversiones y puede permanecer activa durante una sesión con PCW o con MPLAB. Esto puede ser útil durante el proceso de depuración de un programa.

3. OPERADORES Y EXPRESIONES

- **Operadores de asignación**

  Una expresión de asignación tradicional es de la forma

  expr1 = expr1 operador expr2 , es decir, i = i + 5. Esta expresión se puede representar por otra forma más corta: ***expr1 operador= expr2***
  siguiendo con el mismo ejemplo i += 5.

  Es en las expresiones complejas, y no en una tan simple como la del ejemplo, donde se puede apreciar la conveniencia de usar esta notación. La siguiente tabla resúme los operadores de asignación compuesta y su significado.

Operador	Descripción
+=	Asignación de suma
-=	Asignación de resta
*=	Asignación de multiplicación
/=	Asignación de división
%=	Asignación de resto de división
<<=	Asignación de desplazamiento a la izquierda
>>=	Asignación de desplazamiento a la derecha
&=	Asignación de AND de bits
|=	Asignación de OR de bits
^=	Asignación de OR exclusivo de bits
~=	Asignación de negación de bits

- **Operadores aritméticos**

  Los operadores aritméticos se usan para realizar operaciones matemáticas. Se listan en la siguiente tabla:

Operador	Descripción	Ejemplo
+	Suma (enteros o reales)	resul = var1 + var2
-	Resta (enteros o reales)	resul = var1 - var2
*	Multiplicación (enteros o reales)	resul = var1 * var2
/	División (enteros o reales)	resul = var1 / var2
-	Cambio de signo en enteros o reales	-var1
%	Módulo; resto de una división entera	rango = n [A1]% 256

- **Operadores relacionales**

Su misión es comparar dos operandos y dar un resultado entero:
1 (verdadero); 0 (falso).
La siguiente tabla ilustra estos operadores:

Operador	Descripción
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
¡=	Distinto de

- **Operadores lógicos**

Al igual que los operadores relacionales, éstos devuelven 1 (verdadero), 0 (falso) tras la evaluación de sus operandos. La tabla siguiente ilustra estos operadores.

Operador	Descripción
!	NO lógico
&&	Y lógico
||	O lógico

- **Operadores de manejo de bits**

Estos operadores permiten actuar sobre los operandos a nivel de bits y sólo pueden ser de tipo entero (incluyendo el tipo char). Son los que siguen:

Operador	Descripción
~	Negación de bits (complemento a 1)
&	Y de bits (AND)
^^	O exclusivo de bits (EXOR)
|	O de bits (OR)

- **Operadores de incremento y decremento**

Aunque estos operadores forman parte del grupo de operadores de asignación, he preferido separarlos en aras a una mayor claridad. Su comportamiento se asemeja a las instrucciones de incremento incf f,d del ensamblador del µcontrolador PIC 16x84 o inc variable del Intel 8051.

Operador	Descripción
++	Incremento
--	Decremento

- **Operadores de desplazamiento de bits**

Los operadores de desplazamiento otorgan al C capacidad de control a bajo nivel similar al lenguaje ensamblador. Estos operadores utilizan dos operandos enteros (tipo int): el primero es el elemento a desplazar y el segundo, el número de posiciones de bits que se desplaza. Se resúmen en la siguiente tabla:

Operador	Descripción
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

- **Operadores de dirección (&) e indirección (*)**

Los operadores & y * se utilizan para trabajar con punteros (véase tema 11). El lenguaje C está muy influenciado por el uso de punteros. Un puntero es una variable que contiene la dirección de una variable o de una función, es decir, es una variable que apunta a otra variable. Los punteros permiten la manipulación indirecta de datos y códigos. Disponemos de dos operadores; véase la siguiente tabla:

Operador	Descripción
&	De dirección
*	De indirección

El operador de dirección &, nos da la dirección de memoria de su operando. El resultado es un puntero al objeto, esto es, a un tipo de datos. Por ejemplo, si queremos guardar en el puntero p la dirección de memoria de la variable entera contador, debemos hacer lo siguiente:

```
p = &contador;          /* p apunta a la dirección de contador */
```

El operador de indirección *, nos da el valor o contenido de la variable cuya dirección está apuntada por el puntero.

```
p = &contador;          /* p apunta a la dirección de contador */
a = *p;                 /* guarda en a el contenido de la var. apuntada por p */
```

- **Expresiones**

Constantes	
123	Decimal
0123	Octal
0x123	Hex
0b010010	Binario
'x'	Carácter
'\010'	Carácter octal
'\x'	Carácter especial; x puede ser: ¿n,t,b,r,f, ', \d,v?
"abcdef"	Cadena (el carácter nulo se agrega al final)

Identificadores	
ABCDE	Hasta 32 caracteres (no puede empezar con números)
ID[X]	Un subíndice
ID[X][X]	Múltiples subíndices
ID.ID	Referencia a una estructura o una unión
ID-->ID	Referencia a una estructura o una unión

Expresiones en orden descendente de precedencia					
(expr)					
!expr	~expr	++expr	expr++	-expr	expr-
(type)expr	*expr	&value	sizeof(type)		
expr*expr	expr/expr	expr%expr			
expr+expr	expr-expr				
expr<<expr	expr>>expr				
expr<expr	expr<=expr	expr>expr	expr>=expr		
expr==expr	expr!=expr				
expr&expr					
expr^expr					
expr | expr					
expr&& expr					
expr || expr					
!value ? expr: expr					
value=expr	value+=expr	value-=expr			
value*=expr	value/=expr	value%=expr			
value>>=expr	value<<=expr	value&=expr			
value^=expr	value|=expr	expr,expr			

4. DIRECTIVAS DEL PREPROCESADOR

Todas las directivas del pre-procesador comienzan con el caracter # seguido por un comando específico. Algunas de estas directivas son extensiones del C estandar. El C proporciona una directiva del preprocesador, *que los compiladores aceptan*, y que permite ignorar o actuar sobre los datos que siguen. Nuestro compilador admite cualquier directiva del pre-procesador que comienze con ***PRAGMA***, lo que nos asegura la compatibilidad con otros compiladores.

Ejemplo:

```
#INLINE                          /* Estas dos líneas son válidas */
#PRAGMA INLINE
```

A continuación se describen todas y cada una de las directivas del compilador que utilizaremos para programar los microcontroladores PIC.

4.1 CONTROL DE MEMORIA

- **#ASM**
  #ENDASM

Las líneas entre *#ASM* y *#ENDASM* se tratan como código ensamblador. La variable predefinida _RETURN_ puede utilizarse para asignar un valor de retorno a la función desde el código en ensamblador. Tener presente que cualquier código C después de ^I^B #ENDASM ^i^b y antes del final de la función puede falsear el valor de retorno. La sintaxis de los comandos en ensamblador se describen en la tabla 1.

Ejemplo:

```
int paridad (int dato)  {
int contador;

#asm

        movlw  8
        movwf  contador
        movlw  0
        lazo:
        xorwf  dato,w
        rrf    dato,f
        decfsz contador,f
        goto   lazo
        movwf  _return_
 #endasm
}
```

Resumen de las instrucciones en ensamblador	
ADDWF f,d	SUBWF f,d
ANDWF f,d	SWAPF f,d
CLRF f	XORWF f,d
CLRW	BCF f,b
COMF f,d	BSF f,b
DECF f,d	BTFSC f,b
DECFSZ f,d	BTFSS f,b
INCF f,d	ANDLW k
INCFSZ f,d	CALL k
IORWF f,d	CLRWDT
MOVF f,d	GOTO k
MOVPHW	IORLW k
MOVPLW	MOVLW k
MOVWF f	RETLW k
NOP	SLEEP
RLF f,d	XORLW
RRF f,d	OPTION
TRIS k	
Solo PCM	
ADDLW k	RETFIE
SUBLW k	RETURN

- **Nota:**
  - **f** Þ Operando fuente;puede ser una constante, un registro o una variable
  - **d** Þ Operando destino;puede ser una constante (0 o 1) y también W o F
  - **b** Þ Campo que referencia la posición de un bit dentro de un registro
  - **k** Þ Operando inmediato o literal; puede ser una expresión constante

- ***#BIT identificador = x.y***

Esta directiva creará un identificador "id" que puede utilizarse como cualquier SHORT INT (entero corto; un bit). El identificador referenciará un objeto en la posición de memoria **x** más el bit de desplazamiento **y**.

Ejemplos:

```
#bit tiempo = 3.4
int resultado;
#bit resultado_impar = resultado.0
…
if (resultado_ impar)
…
```

- ***#BYTE Identificador = X***

Esta directiva creará un identificador "id" que puede utilizarse como cualquier NT (un byte). El identificador referenciará un objeto en la posición de memoria x, donde x puede ser una constante u otro identificador. Si x es otro identificador, entonces éste estará localizado en la misma dirección que el identificador "id".

Ejemplos:

```
#byte  status = 3
#byte  port_b = 6
struct  {
short int r_w;
short int c_d;
int no_usado : 2;
int dato    : 4; } port_a;
#byte port_a = 5
...

port_a.c_d = 1;
```

- ***#RESERVE***

Permite reservar posiciones de la RAM para uso del compilador.

#RESERVE debe aparecer después de la directiva #DEVICE, de lo contrario no tendrá efecto.

Ejemplo:

```
#RESERVE  0x7d, 0x7e, 0x7f
```

- ***#ROM***

Esta directiva permite insertar datos en el archivo .HEX. En particular, se puede usar para programar la EEPROM de datos de la serie 84 de PIC.

Ejemplo:

```
#rom  0x2100={1,2,3,4,5,6,7,8}
```

- ***#ZERO_RAM***

Directiva que pone a cero todos los registros internos que pueden usarse para mantener variables, antes de que comienze la ejecución del programa.

Y si en lugar de salidas son entradas ¿cómo se pone: #USE FIXED_IO (puerto_INPUTS=pin_x#, pin_x#...) ¿Puede ser PSP???

4.2 CONTROL DEL COMPILADOR

- ***#CASE***

Hace que el compilador diferencie entre mayúsculas y minúsculas. Por defecto el compilador hace esta distinción.

- ***#OPT n***

Esta directiva sólo se usa con el paquete PCW y, establece el nivel de optimización. Se aplica al programa entero y puede aparecer en cualquier parte del archivo fuente. El nivel de optimización 5 es el nivel para los compiladores DOS. El valor por defecto para el compilador PCW es 9 que proporciona una optimización total.

- ***#PRIORITY***

Esta directiva se usa para establecer la prioridad de las interrupciones.
Los elementos de mayor prioridad van primero.

Ejemplo:

```
#priority rtcc,rb          /* la interrupción rtcc ahora tiene mayor prioridad */
```

4.3 IDENTIFICADORES PREDEFINIDOS

- ***_ _DATE_ _***

Este identificador del pre-procesador contiene la fecha actual (en tiempo de compilación) en el formato siguiente: "30-SEP-98"

Ejemplo:

```
printf("Este software fué compilado el día ");
printf(__DATE__);
```

- ***_ _DEVICE_ _***

Este identificador del pre-procesador es definido por el compilador con el número base del dispositivo actual. El número base normalmente es el número que sigue a la/s letra/s en el número de componente o referencia de un dispositivo. Por ejemplo los PIC16C84 tienen el número base 84.

Ejemplo:

```
#if __device__==84
setup_port_a( todas_entradas );
#endif
```

- ***_ _PCB_ _***

Se utiliza para determinar si es el compilador PCB el que está haciendo la compilación.

Ejemplo:

```
#ifdef __pcb__    /* este compilador sólo es válido para PIC cuyo opcode es de 12 bits */
#device PIC16c54
#endif
```

- ***_ _PCM_ _***

Se utiliza para determinar si es el compilador PCM el que está haciendo la compilación.

Ejemplo:

```
#ifdef __pcm__
#device PIC16c71
#endif
```

4.4 DIRECTIVAS DEL C ESTÁNDAR

- ***#DEFINE Identificador CADENA***

Se utiliza simplemente para reemplazar el IDentificador (ID) con CADENA

Ejemplo:

```
#define  BITS  8
#define rotar(x)  (x<<4)

a=a+BITS;                          // Es lo mismo que  a=a+8;
a=rotar(a);                        // Es lo mismo que a=(a<<4);
```

- ***#IF expresión_constante***
  #ELSE
  #ENDIF

El pre-procesador evalúa la expresión_constante y si es distinta de cero procesará las líneas hasta el #ELSE -que es opcional- o en su defecto hasta el #ENDIF.

Ejemplo:

```
#if  (a + b + c) > 8
printf(" Demasiados parámetros ");
#endif
```

- ***#IFDEF***
  #ELSE
  #ENDIF

Esta directiva actúa como el #IF sólo que aquí el pre-procesador simplemente comprueba que reconoce el id especificado (creado con un #DEFINE). Nótese que #IFDEF verifica si se definió un id pero #IFNDEF comprueba que no está definido el id.

Ejemplo:

```
#ifdef DEBUG
printf("punto de debug en ...");
#endif
```

- ***#ERROR***

Esta directiva para el compilador y emite el mensaje que se incluye a continuación (en la misma línea)de la propia directiva. El mensaje puede incluir macros. También puede utilizarse para alertar al usuario de una situación anómala en tiempo de compilación.

Ejemplos:

```
#if  BUFFER_SIZE > 16
#error El tamaño del buffer es demasiado grande
#endif
#error   Macro test:  min(x,y)
```

- **#INCLUDE <Nombre_Fichero>**
  #INCLUDE "Nombre_Fichero"

Esta directiva hace que el compilador incluya en el fichero fuente el texto que contiene el archivo especificado en <Nombre_Fichero>.

Si el nombre del fichero se incluye entre los símbolos '< >' el compilador busca el fichero en el directorio INCLUDE.

Si se pone entre comillas dobles " " el compilador busca primero en el directorio actual o directorio de trabajo y si no lo encuentra, entonces lo busca en los directorios INCLUDE del compilador.

Ejemplo:

#include <16C54.H>
#include "reg_C84.h"

- ***#LIST***

Guarda el código fuente en el archivo .LST

- ***#NOLIST***

No guarda el código fuente en el archivo .LST

- ***#PRAGMA comando***

Esta directiva se usa para mantener compatibilidad entre los compiladores de C. El compilador aceptará esta directiva antes de cualquier otro comando del pre-procesador.

Ejemplo:

#pragma device PIC16C54

- ***#UNDEF Identificador***

El identificador ID no tendrá ya significando para el pre-procesador.

4.5 ESPECIFICACIÓN DE DISPOSITIVOS

- ***#DEVICE CHIP***

Esta directiva define al compilador la arquitectura hardware utilizada. Esto determina la memoria RAM y ROM así como el juego de instrucciones. Para los chips (uC's, memorias, etc) con más de 256 bytes de RAM se puede seleccionar entre punteros de 8 o 16 bits. Para usar punteros de 16 bits hay que añadir *=16 después del nombre del chip (uC, memoria, ...) o en una nueva línea después de la declaración del chip. Se puede obtener informaciçon sobre un dispositivo con el programa PICCHIPS.

Ejemplos:

*#device PIC16C67 *=16*
#device PIC16C74
*#device *=16*

- ***#ID***
  #ID número, número, número
  #ID "nombre_archivo"
  #ID CHECKSUM

Esta directiva define la palabra de IDdentificación que se grabará en el chip (uC, memoria, etc). Esta directiva no afecta a la compilación pero la información se pone en el archivo de salida. La primera sintaxis necesita un número de 16-bit y pondrá un nible en cada una de las cuatro palabras del ID. La segunda sintaxis especifica el valor exacto en cada una de las cuatro palabras del ID. Cuando se especifica "nombre_archivo" el ID se lee del archivo indicado; su formato debe ser texto simple con un CR/LF al final. La palabra CHECKSUM indica que el checksum del dispositivo debe tomarse como el ID.

Ejemplo:

#id 0x1234
#id "NumSerie.txt"
#id CHECKSUM

- ***#FUSES opciones***

Esta directiva define qué fusibles deben activarse en el dispositivo cuando se programe. Esta directiva no afecta a la compilación; sin embargo, esta información se pone en el archivo de salida. Si los fusibles necesitan estar en formato Parallax, hay que agregar PAR en opciones. Utilizar la utilidad PICCHIPS para determinar qué opciones son válidas para cada dispositivo. La opción SWAP tiene la función especial de intercambiar, los bytes alto y bajo de los datos que no son parte del programa, en el archivo Hex. Esta información es necesaria para algunos programadores de dispositivos. Algunas de las opciones más usadas son:

LP, XT, HS, RC
WDT, NOWDT
PROTECT, NOPROTECT
PUT, NOPUT (Power Up Timer)
BROWNOUT, NOBROWNOUT
PAR (Parallax Format Fuses)
SWAP

Ejemplo:

#fuses HS,WDT

4.6 CALIFICADORES DE FUNCIÓN

- ***#INLINE***

Esta directiva le dice al compilador que el procedimiento que sigue a la directiva será llevado a cabo EN LÍNEA. Esto causará una copia del código que será puesto en cualquier parte donde se llame al procedimiento. Esto es útil para ahorrar espacio de la pila (stack) y aumentar la velocidad.

Sin esta directiva es el compilador quien decidirá cuando es mejor hacer los procedimientos EN LÍNEA.

Ejemplo:

```
#inline
swap_byte(int &a, int &b)
  {
int t;
t=a;
a=b;
b=t;
  }
```

- ***#INT_DEFAULT función_de_interrupción_por_defecto***

La función que sigue a la directiva será llamada si el PIC activa una interrupción y ninguno de los flags de interrupción está activo.

Ejemplo:

```
#int_default

control_interrupcion()
  {

        activar_int=VERDADERO;

  }
```

- ***#INT_GLOBAL función***

La función que sigue a esta directiva reemplaza al distribuidor de interrupciones del compilador; dicha función toma el control de las interrupciones y el compilador no salva ningún registro. Normalmente no es necesario usar esto y debe tratarse con gran prudencia.

- ***#INT_xxx función_de_interrupción***

Estas directivas especifican que la función que le sigue es una función de interrupción. Las funciones de interrupción no pueden tener ningún parámetro. Como es natural, no todas las directivas pueden usarse con todos los dispositivos. Las directivas de este tipo que disponemos son:

#INT_EXT	INTERRUPCIÓN EXTERNA
#INT_RTCC	DESBORDAMIENTO DEL TIMER0(RTCC)
#INT_RB	CAMBIO EN UNO DE LOS PINES B4,B5,B6,B7
#INT_AD	CONVERSOR A/D
#INT_EEPROM	ESCRITURA EN LA EEPROM COMPLETADA
#INT_TIMER1	DESBORDAMIENTO DEL TIMER1
#INT_TIMER2	DESBORDAMIENTO DEL TIMER2
#INT_CP1	MODO CAPTURA DE DATOS POR CCP1
#INT_CCP2	MODO CAPTURA DE DATOS POR CCP2
#INT_SSP	PUERTO DE SERIE INTELIGENTE(SPI, I2C)
#INT_PSP	PUERTO PARALELO
#INT_TBE	SCI DATO SERIE TRANSMITIDO
#INT_RDA	SCI DATO SERIE RECIBIDO
#INT_COMP	COMPARADOR DE INTERRUPCIONES
#INT_ADOF	DESBORDAMIENTO DEL A/DC DEL PIC 14000
#INT_RC	CAMBIO EN UN PIN Cx
#INT_I2C	I2C DEL 14000
#INT_BUTTON	PULSADOR DEL 14000
#INT_LCD	LCD 92x

El compilador salta a la función de interrupción cuando se detecta una interrupción. Es el propio compilador el encargado de generar el código para guardar y restaurar el estado del procesador.

También es el compilador quien borrará la interrupción (el flag). Sin embargo, nuestro programa es el encargado de llamar a la función ENABLE_INTERRUPT() para activar previamente la interrupción junto con el señalizador (flag) global de interrupciones.

Ejemplo:

```
#int_ad

control_adc()
  {

        adc_activo=FALSO;

                                                    }
```

- ***#SEPARATE función***

#SEPARATE le dice al compilador que el procedimiento o función que sigue a la directiva será llevado a cabo por SEPARADO. Esto es útil para evitar que el compilador haga automáticamente un procedimiento en línea (INLINE). Esto ahorra memoria ROM pero usa más espacio de la pila. El compilador hará todos los procedimientos #SEPARATE, separados, tal como se solicita, aun cuando no haya bastante pila.

Ejemplo:

```
#separate
swap_byte (int *a, int *b) {
      int t;
      t=*a;
      *a=*b;
      *b=t;
}
```

4.7 LIBRERÍAS INCORPORADAS

- ***#USE DELAY (CLOCK=frecuencia)***

Esta directiva indica al compilador la frecuencia del procesador, en ciclos por segundo, a la vez que habilita el uso de las funciones DELAY_MS() y DELAY_US(). Opcionalmente podemos usar la función restart_WDT() para que el compilador reinicie el WDT durante el retardo.

Ejemplos:

#use delay (clock=20000000)
#use delay (clock=32000, RESTART_WDT)

- ***#USE FAST_IO (puerto)***

Esta directiva afecta al código que el compilador generará para las instrucciones de entrada y salida. Este método rápido de hacer I/O ocasiona que el compilador realice I/O sin programar el registro de dirección. El puerto puede ser A-G.

Ejemplo:

#use fast_io(A)

- ***#USE FIXED_IO (puerto_OUTPUTS=pin_x#, pin_x#...)***

Esta directiva afecta al código que el compilador generará para las instrucciones de entrada y salida. El método fijo de hacer I/O causará que el compilador genere código para hacer que un pin de I/O sea entrada o salida cada vez que se utiliza. Esto ahorra el byte de RAM usado en I/O normal.

Ejemplo:

#use fixed_io(a_outputs=PIN_A2 ,PIN_A3)

- ***#USE I2C (master/slave, SDA=Pin, SCL=Pin opciones)***

La librería I2C contiene funciones para implementar un bus I2C. La directiva **#USE I2C** permanece efectiva para las funciones **I2C_START, I2C_STOP, I2C_READ, I2C_WRITE e I2C_POLL** hasta que se encuentre otra directiva **#USE I2C**.

Se generan las funciones software a menos que se especifique la opción NOFORCE_SW. El modo SLAVE sólo debe usarse con las funciones SSP. Las opciones son:

OPCIONES:

MASTER	Establece el modo maestro o principal
SLAVE	Modo esclavo
SCL=pin	Especifica el pin SCL (es un bit de dirección)
SDA=pin	Especifica el pin SDA
ADDRESS=nn	Especifica la dirección del modo esclavo
FAST	Usa la especificación rápida I2C
SLOW	Usa la especificación lenta I2C
RESTART_WDT	Reinicia el WDT mientras espera en I2C_READ
NOFORCE_SW	Usa funciones hardware I2C

Ejemplos:

#use I2C(master, sda=PIN_B0, scl=PIN_B1)
#use I2C(slave,sda=PIN_C4,scl=PIN_C3 address=0xa0,NOFORCE_SW)

- ***#USE RS232 (BAUD=baudios, XMIT=pin, RCV=pin...)***

Esta directiva le dice al compilador la velocidad en baudios y los pines utilizados para la I/O serie. Esta directiva tiene efecto hasta que se encuentra otra directiva RS232.

La directiva **#USE DELAY** debe aparecer antes de utilizar **#USE RS232**. Esta directiva habilita el uso de funciones tales como GETCH, PUTCHAR y PRINTF. Si la I/O no es estandar es preciso poner las directivas **FIXED_IO** o **FAST_IO** delante de **#USE RS232**

OPCIONES:

RESTART_WDT	Hace que GETC() ponga a cero el WDT mientras espera un carácter.
INVERT	Invierte la polaridad de los pines serie (normalmente no es necesario con el convertidor de nivel, como el MAX232). No puede usarse con el SCI interno.
PARITY=X	Donde X es N, E, u O.
BITS =X	Donde X es 5-9 (no puede usarse 5-7 con el SCI).
FLOAT_HIGH	Se utiliza para las salidas de colecctor abierto.
ERRORS	Indica al compilador que guarde los errores recibidos en la variable RS232_ERRORS para restablecerlos cuando se producen.

BRGH1OK	Permite velocidades de transmisión bajas en chips (uC's, memorias, etc) que tienen problemas de transmisión. Cuando utilizamos dispositivos con SCI y se especifican los pines SCI, entonces se usará el SCI. Si no se puede alcanzar una tasa de baudios dentro del 3% del valor deseado utilizando la frecuencia de reloj actual, se generará un error.
ENABLE=pin	El pin especificado estará a nivel alto durante la transmisión.
FORCE_SW	Usa una UART software en lugar del hardware aun cuando se especifican los pines del hardware. La definición de RS232_ERRORS es como sigue: **Sin UART:** El bit 7 es el 9º bit para el modo de datos de 9 bit. El bit 6 a nivel alto indica un fallo en el modo flotante alto. **Con UART:** Usado sólo para conseguir: Copia del registro RCSTA, excepto: que el bit 0 se usa para indicar un error de paridad. **Ejemplo:** *#use rs232(baud=9600, xmit=PIN_A2,rcv=PIN_A3)*

- ***#USE STANDARD_IO (puerto)***

Esta directiva afecta al código que el compilador genera para las instrucciones de entrada y salida. El método standar de hacer I/O causará que el compilador genere código para hacer que un pin de I/O sea entrada o salida cada vez que se utiliza. En los procesadores de la serie 5X esto necesita un byte de RAM para cada puerto establecido como I/O estandar.

Ejemplo:

#use standard_io(A)

5. FUNCIONES PERMITIDAS POR EL COMPILADOR DE C

5.1 FUNCIONES DE I/O SERIE RS232

- ***c = GETC()***
  c = GETCH()
  c = GETCHAR()

Estas funciones esperan un carácter por la patilla RCV del dispositivo RS232 y retorna el carácter recibido.

Es preciso utilizar la directiva #USE RS232 antes de la llamada a esta función para que el compilador pueda determinar la velocidad de transmisión y la patilla utilizada. La directiva #USE RS232 permanece efectiva hasta que se encuentre otra que anule la anterior.

Los procedimientos de I/O serie exigen incluir #USE DELAY para ayudar a sincronizar de forma correcta la velocidad de transmisión. Recordad que es necesario adaptar los niveles de voltaje antes de conectar el PIC a un dispositivo RS-232.

Ejemplo:

```
printf("Continuar (s,n)?");
do {
      respuesta=getch();
   } while(respuesta!='s'&& respuesta!='n');
```

- ***GETS(char *string)***

Esta función lee caracteres (usando GETC()) de la cadena (string) hasta que encuentra un retorno de carro(valor ASCII 13). La cadena se termina con un 0.

Ejemplo:

Véase la función GET_STRING en el archivo INPUT.C.

- ***PUTC()***
  PUTCHAR()

Estas funciones envían un carácter a la patilla XMIT del dispositivo RS232. Es preciso utilizar la directiva #USE RS232 antes de la llamada a esta función para que el compilador pueda determinar la velocidad de transmisión y la patilla utilizada. La directiva #USE RS232 permanece efectiva hasta que se encuentre otra que anule la anterior.

Ejemplo:

```
if (checksum==0)
        putchar(ACK);
else
putchar(NAK);                    // NAK carácter de respuesta negativa
```

- ***PUTS(string)***

Esta función envía cada carácter de string a la patilla XMIT del dispositivo RS232. Una vez concluido el envío de todos los caracteres la función envía un retorno de carro CR o RETURN (ASCII 13) y un avance de línea LF o LINE-FEED (ASCII 10).

Ejemplo:

```
puts( " ---------- " );
puts( " |  HOLA   |" );
puts( " ---------- " );
```

- ***PRINTF([function], string, [values])***

La función de impresión formateada PRINTF saca una cadena de caracteres al estándar serie RS-232 o a una función especificada. El formato está relacionado con el argumento que ponemos dentro de la cadena (string).

Cuando se usan variables, string debe ser una constante. El carácter % se pone dentro de string para indicar un valor variable, seguido de uno o más caracteres que dan formato al tipo de información a representar.

Si ponemos %% obtenemos a la salida un solo %. El formato tiene la forma genérica %wt, donde w es optativo y puede ser 1,2,...,9. Esto es para especificar cuántos carácteres son representados; si elegimos el formato 01,...,09 indicamos ceros a la izquierda, o también 1.1 a 9.9 para representación en punto flotante.

t es el tipo de formato y puede ser uno de los siguientes:

C	Carácter
U	Entero sin signo
x	Entero en Hex (en minúsculas)
X	Entero en Hex (en mayúsculas)
D	Entero con signo
%e	Real en formato exponencial(notación científica)
%f	Real (Float)
Lx	Entero largo en Hex (en minúsculas)
LX	Entero largo en Hex (en mayúsculas)
Lu	Decimal largo sin signo
Ld	Decimal largo con signo
%	Simplemente un %

Ejemplos:

```
byte  x,y,z;
printf (" Hola ");
printf("RTCCValue=>%2x\n\r",get_rtcc());
printf("%2u %X %4X\n\r",x,y,z);
printf(LCD_PUTC, "n=%c",n);
```

Ejemplos de formatos:

Especificador	Valor=0x12	Valor=0xfe
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2[A1]
%x	12	fe
%X	12	FE
%4X	0012	00FE

* El resultado es impreciso - Información no válida.

- ***KBHIT()***

Esta función devuelve TRUE si el bit que se está enviando al pin RCV de un dispositivo RS232, es el bit de inicio de un carácter. Es preciso utilizar la directiva #USE RS232 antes de la llamada a esta función para que el compilador pueda determinar la velocidad en baudios y la patilla utilizada. La directiva #USE RS232 permanece efectiva hasta que se encuentre otra que anule la anterior.

Ejemplo:

```
keypress=' ';
while ( keypress!='Q' ) {          // entramos al bucle while
  if ( kbhit () )
     keypress=getc();              // en la variable keypress se guardan los caracteres
  if (!input(PIN_B2))              // inicio del envío de un byte
        output_high(PIN_B3);
  else
        output_low(PIN_B3)
}
```

- ***SET_UART_SPEED(baud)***

Esta función cambia la velocidad de transmisión de la UART (Universal Asynchronous Receiver Transmitter) en tiempo de ejecución.

- ***SETUP_ADC(mode)***

Esta función configura (permite establecer los parámetros) del conversor analógico/digital. Para el chip 14000, esta función establece la corriente de carga. Los modos son los siguientes:

ADC_OFF
ADC_CLOCK_DIV_2
ADC_CLOCK_DIV_8
ADC_CLOCK_DIV_32
ADC_CLOCK_INTERNAL

Ejemplo:

```
setup_adc( ADC_CLOCK_INTERNAL );
```

5.2 FUNCIONES DE I/O CON EL BUS I2C

- ***b = I2C_POLL()***

Esta función retorna un valor distinto de cero (TRUE) cuando el hardware ha recibido un byte en el buffer. En ese momento se produce una llamada a la función I2C_READ () que devolverá inmediatamente el byte recibido. I2C_POLL() está disponible sólo cuando se usa el SSP.

Ejemplo:

```
i2c_start();              // condición de inicio
i2c_write(0xc1);          // direccionamiento/lectura del dispositivo
count=0;
while(count!=4) {
f(i2c_poll())
r[count++]= i2c_read();
        // leer próximo byte
        // tratamiento de la información
}
i2c_stop();       // condición de parada
```

- ***I2C_READ()***

La función I2C_READ() lee un byte del interface I2C. Es necesario especificar la directiva #USE I2C antes de la llamada a I2C_READ().

En modo 'master' esta función genera los impulsos de reloj y en modo 'esclavo' permanece a la espera de estos impulsos. Cuando la función espera los datos no se puede producir ninguna interrupción.

Si incluimos la opción RESTART_WDT en la directiva #USE I2C entonces esta función activa el WDT o perro guardián mientras está esperando. Se puede utilizar el parámetro optativo '0' para que la función no envíe el carácter acuse de recibo (ACK), de los datos recibidos.

Ejemplo:

```
i2c_start();          // condición de inicio
i2c_write(0xa1);      // dirección del dispositivo
r1 = i2c_read();      // Lee el byte primero
r2 = i2c_read();      // Lee segundo byte
i2c_stop();           // condición de paro
```

- ***I2C_START()***

Esta función lanza una condición de inicio cuando el dispositivo I2C está modo master. Es necesario especificar la directiva #USE I2C antes de la llamada a I2C_START().

Después de la condición de inicio el reloj se maniene a nivel bajo hasta que se llama a las funciones I2C_READ() e I2C_WRITE().

Ejemplo:

Véase la función I2C_WRITE.

- ***I2C_STOP()***

Esta función lanza una condición de stop o paro cuando el dispositivo I2C está modo master. Hay que especificar la directiva #USE I2C antes de la llamada a I2C_STOP()

Ejemplo:

Véase la función I2C_WRITE.

- ***I2C_WRITE(byte)***

La función I2C_WRITE() envia un byte al interface I2C. Hay que especificar la directiva #USE I2C antes de la llamada a I2C_WRITE(). En modo 'master' la propia función genera la señal de reloj con los datos y en modo 'esclavo' esperará la señal de reloj del 'master'. Esta función devuelve el Bit de acuse de recibo (ACK).

Ejemplo:

```
i2c_start();          // condición de inicio
i2c_write(0xa0);      // dirección del dispositivo
i2c_write(5);         // envío de una orden al dispositivo
i2c_write(12);        // envío de datos al dispositivo
i2c_stop();           // condición de parada
```

5.3 FUNCIONES DE I/O DISCRETA

- ***INPUT(pin)***

Devuelve el estado '0' o '1' de la patilla indicada en pin. El método de acceso de I/O depende de la última directiva #USE *_IO utilizada. El valor de retorno es un entero corto.

Ejemplo:

while (!input(PIN_B1));

Nota: El argumento para las funciones de entrada y salida es una dirección de bit. Por ejemplo, para el bit 3º del port A (byte 5 de los SFR) tendría un valor dirección de 5*8+3=43.
Esto se puede definir como sigue: *#define pin3_portA 43.*

Los pines o patillas de los dispositivos estan definidos como PIN_XX en los archivos de cabecera *.H. Éstos, se pueden modificar para que los nombres de los pines sean más significativos para un proyecto determinado.

- ***OUTPUT_BIT(pin, value)***

Esta función saca el bit dado en value(0 o 1) por la patilla de I/O especificada en pin. El modo de establecer la dirección del registro, está determinada por la última directiva #USE *_IO.

Ejemplo:

```
output_bit( PIN_B0, 0);                    // es lo mismo que output_low(pin_B0);
output_bit( PIN_B0,input( PIN_B1 ) );      // pone B0 igual que B1
output_bit( PIN_B0, shift_left(&data, 1, input(PIN_B1)));
                    // saca por B0 el MSB de 'data' y al mismo tiempo
                    // desplaza el nivel en B1 al LSB de data.
```

- ***OUTPUT_FLOAT(pin)***

Esta función pone la patilla especificada como pin en el modo de entrada. Esto permitirá que la patilla esté flotante para representar un nivel alto en una conexión de tipo colector abierto.

Ejemplo:

```
                                   // guardamos la lectura del port A en dato
if( (dato & 0x80)==0 )             // comprobamos si es '1' el MSB
        output_low(pin_A0);        // si es '1' ponemos a cero el pin A0
else
        output_float(pin_A0);      // de lo contrario, ponemos el pin A0 a uno
```

- ***OUTPUT_HIGH(pin)***

Pone a 'uno' el pin indicado. El método de acceso de I/O depende de la última directiva #USE *_IO utilizada.

Ejemplo:

output_high(PIN_A0);

- ***OUTPUT_LOW(pin)***

Pone a 'cero' el pin indicado. El método de acceso de I/O depende de la última directiva #USE *_IO.

Ejemplo:

output_low(PIN_A0);

- ***PORT_B_PULLUPS(flag)***

Esta función activa/desactiva las resistencias pullups en las entradas del puerto B. Flag puede ser TRUE (activa) o FALSE (desactiva).

Ejemplo:

port_b_pullups(FALSE);

- ***SET_TRIS_A(value)***
  SET_TRIS_B(value)
  SET_TRIS_C(value)
  SET_TRIS_D(value)
  SET_TRIS_E(value)

Estas funciones permiten escribir directamente los registros tri-estado para la configuración de los puertos.

Esto debe usarse con FAST_IO() y cuando se accede a los puertos de I/O como si fueran memoria, igual que cuando se utiliza una directiva #BYTE. Cada bit de value representa una patilla. Un '1' indica que la patilla es de entrada y un '0' que es de salida.

Ejemplo:

```
SET_TRIS_B( 0x0F );    // pone B0, B1, B2 y B3 como entradas; B4, B5, B6 y B7
                       // como salidas, en un PIC 16c84
```

5.4 FUNCIONES DE RETARDOS

- ***DELAY_CYCLES(count)***

Esta función realiza retardos según el número de ciclos de instrucción especificado en count; los valores posibles van desde 1 a 255. Un ciclo de instrucción es igual a cuatro periodos de reloj.

Ejemplo:

```
delay_cycles( 3 );   //  retardo de 3ciclos instrucción; es igual que un NOP
```

- ***DELAY_MS(time)***

Esta función realiza retardos del valor especificado en time. Dicho valor de tiempo es en milisegundos y el rango es 0-65535.

Para obtener retardos más largos así como retardos 'variables' es preciso hacer llamadas a una función separada; véase el ejemplo siguiente.

Es preciso utilizar la directiva #use delay(clock=frecuencia) antes de la llamada a esta función, para que el compilador sepa la frecuencia de reloj.

Ejemplos:

```
#use delay (clock=4000000)              // reloj de 4MHz
delay_ms( 2 );                          // retardo de 2ms

void retardo_segundos(int n) {          // retardo de 'n' segundos; 0 <= n => 255
for (; n!=0; n--)
delay_ms(  1000 );              // 1 segundo
}
```

- ***DELAY_US(time)***

Esta función realiza retardos del valor especificado en time. Dicho valor es en microsegundos y el rango va desde 0 a 65535. Es necesario utilizar la directiva #use delay antes de la llamada a esta función para que el compilador sepa la frecuencia de reloj.

Ejemplos:

```
#use delay(clock=20000000)
delay_us(50);
int espera = 10;
delay_us(espera);
```

5.5 FUNCIONES DE CONTROL DEL PROCESADOR

- ***DISABLE_INTERRUPTS(level)***

Esta función desactiva la interrupción del nivel dado en level. El nivel GLOBAL prohíbe todas las interrupciones, aunque estén habilitadas o permitidas. Los niveles de interrupción son:

o GLOBAL	o INT_AD	o INT_CCP2	o INT_COMP
o INT_EXT	o INT_EEPROM	o INT_SSP	o INT_ADOF
o INT_RTCC	o INT_TIMER1	o INT_PSP	o INT_RC
o INT_RB	o INT_TIMER2	o INT_TBE	o INT_I2C
o INT_AD	o INT_CP1	o INT_RDA	o INT_BUTTON

Ejemplo:

```
disable_interrupts(GLOBAL);    /* prohíbe todas las interrupciones */
```

- ***ENABLE_INTERRUPTS(level)***

Esta función activa la interrupción del nivel dado en level. Queda a cargo del técnico definir un procedimiento o rutina de atención, para el caso que se produzca la interrupción indicada. El nivel GLOBAL permite todas las interrupciones que estén habilitadas de forma individual. Véase también DISABLE_INTERRUPTS.

Ejemplo:

```
disable_interrupts(GLOBAL);     // Prohíbe todas las interrupciones

enable_interrupts(INT_AD);      // Quedan habilitadas estas dos interrupciones,
enable_interrupts(INT_I2C);     //pero hasta que no se habilite GLOBAL, no
                                //podrán activarse

enable_interrupts(GLOBAL);      // Ahora sí se pueden producir las interrupciones
                                //anteriores
```

- ***EXT_INT_EDGE(edge)***

Esta función determina el flanco de activación de la interrupción externa. El flanco puede ser de subida (L_TO_H) o de bajada (H_TO_L).

Ejemplo:

```
ext_int_edge( L_TO_H );
```

- ***READ_BANK(bank, offset)***

Esta función lee un byte de datos del banco de memoria especificado en bank. El banco puede ser 1, 2 o 3 y sólo para la serie '57 de PIC; el desplazamiento u offset está entre 0 y 15.

Ejemplo:

dato = READ_BANK(1,5);

- ***RESTART_CAUSE()***

Esta función devolverá la razón por la que se ha producido el último reset del procesador. Los valores de retorno pueden ser:

WDT_FROM_SLEEP
WDT_TIMEOUT
MCLR_FROM_SLEEP
NORMAL_POWER_UP

Ejemplo:

```
switch ( restart_cause() ) {
case WDT_FROM_SLEEP: ;
case WDT_TIMEOUT:
        handle_error();
}
```

- ***SLEEP()***

Esta función pone al micro en un estado de REPOSO.

Ejemplo:

SLEEP();

- ***WRITE_BANK(bank, offset, value)***

Esta función escribe un byte de datos en el banco de memoria especificado. Value es el byte de datos a escribir; bank puede ser 1-3 según el PIC; offset puede ser 0-15.

Ejemplo:

WRITE_BANK(1, 0, 23);

5.6 CONTADORES / TEMPORIZADORES

- ***GET_RTCC()***
  GET_TIMER0()
  GET_TIMER1()
  i=GET_TIMER2()

Estas funciones devuelven el valor de la cuenta de un contador en tiempo real. RTCC y Timer0 son el mismo. Timer1 es de 16 bits y los otros son de 8 bits.

Ejemplo:

```
while ( get_rtcc() != 0 ) ;
```

- ***RESTART_WDT()***

Esta función reiniciará el timer del watchdog. Si habilitamos el timer del watchdog, debe llamarse periódicamente a RESTART_WDT() para prevenir el reseteo del procesador.

Ejemplo:

```
while (!done) {
restart_wdt();
.
        .
        }
```

- ***SET_RTCC(value)***
  SET_TIMER0(value)
  SET_TIMER1(value)
  SET_TIMER2(value)

Estas funciones activan el timer o temporizador al valor especificado. RTCC y Timer0 son el mismo. Timer1 es de 16 bits y los otros son de 8 bits.

Ejemplo:

```
if ( get_rtcc()==25 ) set_rtcc(0);
```

- ***SETUP_COUNTERS(rtcc_state, ps_state)***

Esta función inicializa el timer RTCC o el WDT. El rtcc_state determina qué es lo que activa el RTCC. El ps_state establece un pre-scaler para el RTCC o el WDT. El pre-scaler alarga el ciclo del contador indicado. Si se activa el pre-scaler del RTCC el WDT se pondrá a WDT_18MS. Si se activa el pre-scaler del WDT el RTCC se pone a RTCC_DIV_1.

Valores del rtcc_state:	RTCC_INTERNAL RTCC_EXT_L_TO_H RTCC_EXT_H_TO_L
Valores del ps_state:	RTCC_DIV_2 RTCC_DIV_4 RTCC_DIV_8 RTCC_DIV_16 RTCC_DIV_32 RTCC_DIV_64 RTCC_DIV_128 RTCC_DIV_256 WDT_18MS WDT_36MS WDT_72MS WDT_144MS WDT_288MS WDT_576MS WDT_1152MS WDT_2304MS

Ejemplo:

setup_counters (RTCC_INTERNAL, WDT_2304MS);

- ***SETUP_TIMER_1(mode)***

Esta función inicializa el timer1. Los valores de mode deben ordenarse juntos, tal como se muestra en el ejemplo. El valor del timer puede leerse y puede escribirse utilizando GET_TIMER1() y SET_TIMER1().

Los valores de mode son:

- T1_DISABLED
- T1_INTERNAL
- T1_EXTERNAL
- T1_EXTERNAL_SYNC
- T1_CLK_OUT
- T1_DIV_BY_1
- T1_DIV_BY_2
- T1_DIV_BY_4
- T1_DIV_BY_8

Ejemplos:

setup_timer_1 (T1_DISABLED);
setup_timer_1 (T1_INTERNAL | T1_DIV_BY_4);
setup_timer_1 (T1_INTERVAL | T1_DIV_BY_8);

- ***SETUP_TIMER_2(mode, period, postscale)***

Esta función inicializa el timer2; mode especifica el divisor del reloj del oscilador. period es un número comprendido entre 0-255, y determina el momento en el que el valor del reloj se resetea a 0. postscale es un número de 0 a 15, que determina cuántos reset del timer se han producido antes de una interrupción. 0 significa 1 reset, 1 significa 2 reset, y así sucesivamente. El valor del timer puede leerse y puede escribirse utilizando GET_TIMER2() y SET_TIMER2().

Los valores de mode son:

- T2_DISABLED
- T2_DIV_BY_1
- T2_DIV_BY_4
- T2_DIV_BY_16

Ejemplo:

setup_timer_2 (T2_DIV_BY_4, 0xc0, 2);

5.7 FUNCIONES DE I/O PSP PARALELA

- ***b = PSP_INPUT_FULL()***
  b = PSP_OUTPUT_FULL()
  b = PSP_OVERFLOW()

Estas funciones verifican el PSP para las condiciones indicadas y devuelven VERDADERO o FALSO.

Ejemplos:

```
while (psp_output_full());
psp_data = command;
while(!psp_input_full());
if ( psp_overflow() )
error = TRUE;
else
data = psp_data;
```

- ***SETUP_PSP(mode)***

Esta función inicializa el PSP; mode puede ser:

- PSP_ENABLED
- PSP_DISABLED

La función SET_TRIS_E(value) puede usarse para establecer la dirección de los datos. Los datos pueden leerse y escribirse utilizando la variable PSP_DATA.

5.8 FUNCIONES DE I/O SPI A DOS HILOS

- ***SETUP_SPI(mode)***

Esta función inicializa el SPI; mode puede ser:

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

Ejemplo:

setup_spi(spi_master | spi_l_to_h | spi_clk_div_16);

- ***SPI_DATA_IS_IN()***

Esta función devuelve TRUE si se han recibido datos en el SPI.

- ***SPI_READ()***

Esta función devuelve un valor leído por el SPI. Cuando SPI_READ() recibe un dato, se temporiza, y la función devuelve el valor leído. Si no hay datos dispuestos, SPI_READ() permanece a la espera.

Ejemplo:

```
if ( spi_data_is_in() )                    // si ha llegado dato
new_data = spi_read();          // lee el dato
```

- ***SPI_WRITE(value)***

Esta función escribe el valor por el SPI.

Ejemplo:

```
spi_write( data_out );
data_in = spi_read();
```

5.9 FUNCIONES PARA EL LCD

- ***LCD_LOAD(buffer_pointer, offset, length);***

Carga *length* bytes del buffer_pointer en el segmento del área de datos del LCD, comenzando en el offset cuyo valor puede ser (0, ... 15).

- ***LCD_SYMBOL(symbol, b7_addr, b6_addr, b5_addr, b4_addr, b3_addr, b2_addr, b1_addr, b0_addr);***

Carga 8 bits en el segmento del área de datos del LCD, con cada dirección del bit especificado. Si el bit 7 de symbol está a '1' el segmento en B7_addr se pone a '1', en otro caso se pone a '0'.

Esto es igualmente cierto para todos los otros bits de symbol. Nótese que B7_addr es un bit de dirección de la RAM del LCD.

Ejemplo:

```
byte CONST DIGIT_MAP[10]={0X90,0XB7,0X19,0X36,0X54,0X50,0XB5,0X24};

#define DIGIT_1_CONFIG
COM0+2, COM0+4, COM0+5, COM2+4, COM2+1, COM1+4, COM1+5

for(i = 1;i <= 9; ++i){
        LCD_SYMBOL(DIGIT_MAP[i],
        DIGIT_1_CONFIG);
        delay_ms(1000);
}
```

- ***SETUP_LCD(mode,prescale,segments);***

Esta función se usa para inicializar al controlador 923/924 del LCD, donde *mode* puede ser:

- LCD_DISABLED
- LCD_STATIC
- LCD_MUX12
- LCD_MUX13
- LCD_MUX14

y puede ser calificado por:

- STOP_ON_SLEEP
- USE_TIMER_1

Además, *prescale* puede valer entre 0 y 15; *segments* pueden ser cualquiera de los siguiente:

- SEGO_4
- SEG5_8
- SEG9_11
- SEG12_15
- SEG16_19
- SEGO_28
- SEG29_31
- ALL_LCD_PINS

5.10 FUNCIONES DEL C ESTÁNDAR

- ***f=ABS(x)***

Calcula el valor absoluto de un entero. Si el resultado no se puede representar, el comportamiento es impreciso. El prototipo de esta función está en el fichero de cabecera stdlib.h

- ***f=ACOS(x)***

Calcula el valor del arco coseno del número real x. El valor de retorno está en el rango [0,pi]radianes. Si el argumento no está dentro del rango [-1,1] el comportamiento es impreciso. El prototipo de esta función está en el fichero de cabecera math.h

- ***f=ASIN(x)***

Obtiene el valor del arco seno del número real x. El valor de retorno está en el rango [-pi/2, pi/2]radianes. Si el argumento no está dentro del rango [-1,1] el comportamiento es impreciso. El prototipo de esta función está en el fichero de cabecera math.h

- ***f=ATAN(x)***

Calcula el valor del arco tangente del número real x. El valor de retorno está en el rango [-pi/2, pi/2]radianes. El prototipo de esta función está en el fichero de cabecera math.h

- ***i=ATOI(char *ptr)***

Esta función convierte la cadena de caracteres apuntada por ptr en un valor de tipo entero. Acepta argumentos en decimal y en hexadecimal. Si el resultado no se puede representar, el comportamiento es indeterminado. El prototipo de esta función está en el fichero de cabecera stdlib.h

- ***i=ATOL(char *ptr)***

Esta función convierte la cadena de caracteres apuntada por ptr en un número entero largo (long). Acepta argumentos en decimal y en hexadecimal. Si el resultado no se puede representar, el comportamiento es indeterminado. El prototipo de esta función está en el fichero de cabecera stdlib.h

- ***f=CEIL(x)***

Obtiene el valor entero más pequeño, mayor que el número real x, es decir, hace un redondeo por exceso del número real x. El prototipo de esta función está en el fichero de cabecera math.h

Ejemplo:

```
float x = 2.871;
num = ceil(x)          //  num = 3
```

- ***f=EXP(x) ^b^i***

Calcula la función exponencial del número real x. Si la magnitud de x es demasiado grande, el comportamiento es impreciso. El prototipo de esta función está en el fichero de cabecera math.h

Ejemplo:

```
float v1, pi = 3.1416;
v1 = exp(pi);
```

- ***f=FLOOR(x)***

Calcula el valor entero más grande, menor que el número real x, es decir, hace un redondeo por defecto del número real x. El prototipo de esta función está en el fichero de cabecera math.h.

Ejemplo:

```
float x = 3.871;
num = floor(x)          //  num = 3
```

- ***c = ISALNUM(char)***
  c = ISALPHA(char)
  c = ISDIGIT(char)
  c = ISLOWER(char)
  c = ISSPACE(char)
  c = ISUPPER(char)
  c = ISXDIGIT(char)

Todas estas funciones manejan cadenas de caracteres y sus prototipos están en el fichero de cabecera ctype.h. Este fichero contiene las siguientes macros:

Cada función devuelve un valor distinto de cero si:

ISALNUM(X)	X es 0..9, 'A'..'Z', o 'a'..'z'
ISALPHA(X)	X es 'A'..'Z' o 'a'..'z'
ISDIGIT(X)	X es '0'..'9'
ISLOWER(X)	X es 'a'..'z'
ISUPPER(X)	X es 'A'..'Z
ISSPACE(X)	X es un espacio
ISXDIGIT(X)	X es '0'..'9', 'A'..'F', o 'a'..'f'

- ***LABS(x)***

Obtiene el valor absoluto del entero largo x. Si el resultado no puede representarse, el comportamiento es indefinido. El prototipo de esta función está en el fichero de cabecera stdlib.h

- ***LOG(x)***

Calcula el logaritmo natural del número real x. Si el argumento es menor o igual que cero o demasiado grande, el comportamiento es impreciso. El prototipo de esta función está en el fichero de cabecera math.h

- ***LOG10(x)***

Calcula el logaritmo decimal o base-diez del número real x. Si el argumento es menor o igual que cero o demasiado grande, el comportamiento es impreciso. El prototipo de esta función está en el fichero de cabecera math.h

- ***MEMCPY(dest, source, n)***

Esta función copia ^B^I n ^b^i bytes desde source a dest en la memoria RAM. Tanto *dest* como *source* deben ser punteros.

Ejemplo:

memcpy(&structA, &structB, sizeof (structA));

memcpy(arrayA, arrayB, sizeof (arrayA));

memcpy(&structA, &databyte, 1);

- ***MEMSET(dest, value, n) ^b^i***

Esta función pone *n* bytes de memoria a partir de *dest* con el valor *value.* *dest* debe ser un puntero.

Ejemplo:

memset(arrayA, 0, sizeof(arrayA));

memset(%structA, 0xff,sizeof(structA));

- ***SQRT(x)***

Obtiene la raíz cuadrada del número real x. Si el argumento es negativo, el comportamiento es indeterminado.

5.11 FUNCIONES DE MANEJO DE CADENAS

Estas funciones están definidas en el archivo de cabecera string.h, que debe incluirse con una directiva #include en el fuente.

Todas las funciones listadas aquí operan con constantes de cadena como parámetro. Antes de utilizar estas funciones, conviene copiar (con STRCPY) una constante de cadena a una cadena en la RAM.

- ***CHAR * STRCAT (char *s1, char *s2)***

Añade una copia de la cadena s2 al final de s1, y devuelve un puntero a la nueva cadena s1.

- ***CHAR * STRCHR (char *s, char c)***

Encuentra la primera coincidencia del carácter c en la cadena s y devuelve un puntero al carácter.

- ***CHAR * STRRCHR (char *s, char c)***

Encuentra la última coincidencia del carácter c en la cadena s y devuelve un puntero al carácter.

- ***SIGNED INT STRCMP (char *s1, char *s2)***

Compara s1 y s2; devuelve -1 si s1<s2, 0 si s1=s2, 1 si s1>s2.

- ***SIGNED INT STRNCMP (char *s1, char *s2, int n)***

Compara un máximo de n caracteres (que no vayan seguidos de 0) de s1 a s2; devuelve -1 si s1<s2, 0 si s1=s2, y 1 si s1>s2.

- ***SIGNED INT STRICMP (char *s1, char *s2)***

Compara s1 y s2 sin hacer distinción entre mayúsculas y minúsculas. Devuelve -1 si s1<s2, 0 si s1=s2, y 1 si s1>s2.

- ***CHAR * STRNCPY (char *s1, char *s2, int n)***

Copia un máximo de n caracteres (que no vayan seguidos de 0) de s2 a s1; si s2 tiene menos de n caracteres, se añaden '0' al final.

- ***INT STRCSPN (char *s1, char * s2)***

Calcula la longitud de la porción inicial mayor de s1, que consiste enteramente de caracteres que no estan en s2.

- ***INT STRSPN (char *s1, char *s2)***

Calcula la longitud de la porción inicial mayor de s1, que consiste enteramente de caracteres que estan en s2.

- ***INT STRLEN (char *s)***

Obtiene la longitud de s1 (excluyendo el carácter '\0').

- ***CHAR * STRLWR (char *s)***

Reemplaza mayúsculas con minúsculas y devuelve un puntero a s.

- ***CHAR * STRPBRK (char *s1, char *s2)***

Localiza la primera coincidencia de cualquier carácter de s2 en s1 y devuelve un puntero al carácter o s1 si s2 es una cadena vacía.

- ***CHAR * STRSTR (char *s1, char *s2)***

Localiza la primera coincidencia de una secuencia de caracteres de s2 en s1 y devuelve un puntero a la secuencia; devuelve null si s2 es una cadena vacía.

- ***CHAR * STRTOK (char *s1, char *s2)***

Encuentra la próxima muestra en s1, delimitada por un carácter de separación de cadena de s2 (que puede ser diferente de una llamada a la otra); devuelve un puntero a él.

La primera llamada inicia, al principio de S1, la búsqueda del primer carácter que no esté contenido en s2, y devuelve NULL si no lo encuentra.

Si no se encuentra, este es el inicio (punto de parida) del primer token (valor de retorno). La Función entonces busca desde allí un carácter contenido en s2.

Si no se encuentra, el token actual se extiende hasta el extremo de s1, y las búsquedas siguientes de un token devolverán null.

Si se encuentra uno, se sobreescribe por '\0' que termina el token actual. La función guarda el puntero la carácter siguiente desde el que se iniciará la próxima búsqueda.

Cada llamada posterior, con 0 como primer argumento, inicia la búsqueda a partir del puntero guardado.

- ***STRCPY(dest, SRC)***

Copia una constante de cadena en la RAM.

Ejemplo:

char string[10];
.
.
strcpy (string, "Hola");

- ***c=TOLOWER(char)***
  c=TOUPPER(char)

Pasa los caracteres de mayúsculas a minúsculas y viceversa. TOLOWER(X) pasa a minúsculas y devuelve 'a'..'z'; TOUPPER(X) pasa a mayúsculas y devuelve 'A'..'Z' El resto de caracteres no sufre ningún cambio.

5.12 VOLTAJE DE REFERENCIA VREF

- ***SETUP_VREF(mode)***

Sólo los PIC de la serie 16c62x pueden usar esta función (véase el archivo de cabecera 16c620.h), donde mode puede ser:

FALSE	(desactivado)
VREF_LOW	VDD*VALUE/24
VREF_HIGH	VDD*VALUE/32 + VDD/4

En combinación con VALUE y opcionalmente con VREF_A2.

Ejemplo:

```
SETUP_VREF (VREF_HIGH | 6);          // Con VDD=5, el voltage es
                                     // (5*6/32) + 5/4 = 2.1875V
```

5.13 FUNCIONES DE ENTRADA A/D

- ***SETUP_ADC_PORTS(value)***

Esta función configura los pines del ADC para que sean analógicos, digitales o alguna combinación de ambos. Las combinaciones permitidas varían, dependiendo del chip.

Las constantes usadas también son diferentes para cada chip. Véase el archivo de cabecera *.h para cada PIC concreto. Las constantes ALL_ANALOG y NO_ANALOGS son válidas para todos los chips.

Algunos otros ejemplos de constantes son:

RA0_RA1_RA3_ANALOG

Esto hace que los pines A0, A1 y A3 sean analógicos y los restantes sean digitales. Los +5v se usan como referencia; véase el siguiente ejemplo:

RA0_RA1_ANALOG_RA3_REF

Las patillas A0 y A1 son analógicas; la patilla RA3 se usa como voltaje de referencia y todas las demás patillas son digitales.

Ejemplo:

Setup_adc_ports(ALL_ANALOG);

- ***SETUP_ADC(mode)***

Esta función prepara o configura el conversor A/D. Para la serie 14000 esta función establece la corriente de carga. Véase el archivo 14000.H para los valores según el modo de funcionamiento. Los modos son:

- ADC_OFF
- ADC_CLOCK_DIV_2
- ADC_CLOCK_DIV_8
- ADC_CLOCK_DIV_32
- ADC_CLOCK_INTERNAL

Ejemplo:

setup_adc(ADC_CLOCK_INTERNAL);

- ***SET_ADC_CHANNEL(canal)***

Especifica el canal a utilizar por la función READ_ADC(). El número de canal empieza en 0. Es preciso esperar un corto espacio de tiempo después de cambiar el canal de adquisición, antes de que se puedan obtener lecturas de datos válidos.

Ejemplo:

set_adc_channel(2);

- ***i=READ_ADC()***

Esta función lee el valor digital del conversor analógico digital. Deben hacerse llamadas a SETUP_ADC() y SET_ADC_CHANNEL() en algún momento antes de la llamada a esta función.

Ejemplo:

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

5.14 FUNCIONES CCP

- ***SETUP_CCP1(mode)***
  SETUP_CCP2(mode)

Estas funciones inicializa el contador CCP. Para acceder a los contadores CCP se utilizan las variables CCP_1 y CCP_2. Los valores para mode son:

CCP_OFF
CCP_CAPTURE_FE
CCP_CAPTURE_RE
CCP_CAPTURE_DIV_4
CCP_CAPTURE_DIV_16
CCP_COMPARE_SET_ON_MATCH
CCP_COMPARE_CLR_ON_MATCH
CCP_COMPARE_INT
CCP_COMPARE_RESET_TIMER
CCP_PWM
CCP_PWM_PLUS_1 (sólo si se utiliza un ciclo de trabajo de 8 bits)
CCP_PWM_PLUS_2 (sólo si se utiliza un ciclo de trabajo de 8 bits)
CCP_PWM_PLUS_3 (sólo si se utiliza un ciclo de trabajo de 8 bits)

- ***SETUP_COMPARATOR(mode)***

Sólo los PIC de la serie 16c62x pueden usar esta función (véase el archivo de cabecera 16c620.h), donde mode puede ser:

A0_A3_A1_A2
A0_A2_A1_A2
NC_NC_A1_A2
NC_NC_NC_NC
A0_VR_A2_VR
A3_VR_A2_VR
A0_A2_A1_A2_OUT_ON_A3_A4
A3_A2_A1_A2

Cada uno de los cuatro ítems separado por '_' son C1-, C1+, C2 -, C2+

Ejemplo:

setup_comparator (A0_A3_A1_A2); //inicializa dos comparadores independientes

- ***SET_PWM1_DUTY(value)***
  SET_PWM2_DUTY(value)

Estas funciones escriben los 10 bits de value al dispositivo PWM para establecer el ciclo de trabajo. Se puede usar un valor de 8 bits si no son necesarios los bits menos significativos.

5.15 FUNCIONES PARA EL MANEJO DE LA EEPROM INTERNA

- ***READ_CALIBRATION(n)***

Esta función lee "n" posiciones de la memoria de calibración de un 14000.

Ejemplo:

Fin = read_calibration(16);

- ***READ_EEPROM(address)***

Esta función lee un byte de la dirección (address) de EEPROM especificada. La dirección puede ser 0-63.

Ejemplo:

```
#define LAST_VOLUME  10
volume = read_EEPROM (LAST_VOLUME );
```

- ***WRITE_EEPROM(address, value)***

Esta función escribe un byte de datos en la dirección de memoria EEPROM especificada. address puede valer 0-63; value es el byte de datos a escribir; Esta función puede tardar varios milisegundos para ejecutarse.

Ejemplo:

```
#define LAST_VOLUME  10
volume++;
write_eeprom(LAST_VOLUME,volume);
```

5.16 FUNCIONES PARA LA MANIPULACIÓN DE BITS

- ***BIT_CLEAR(var,bit)***

Esta función simplemente borra (pone a '0') el dígito especificado en bit(0-7 o 0-15) del byte o palabra aportado en var. El bit menos significativo es el 0.

Esta función es exactamente igual que: var & = ~(1 << bit);

Ejemplo:

```
int x;
x=5;
bit_clear(x,2);          // x = 1
```

- ***BIT_SET(var,bit)***

Esta función pone a '1' el dígito especificado en bit(0-7 o 0-15) del byte o palabra aportado en var. El bit menos significativo es el 0.
Esta función es igual que: var | = (1 << bit);

Ejemplo:

```
int x;
x=5;
bit_set(x,3);            // x = 13
```

- ***BIT_TEST(var,bit)***

Esta función examina el dígito especificado en bit(0-7 o 0-15) del byte o palabra aportado en var. Esta función es igual, aunque mucho más eficaz que esta otra forma: ((var & (1 << bit)) != 0)

Ejemplo:

```
if( bit_test(x,3) || !bit_test (x,1) ){
                         //o el bit 3 es 1 o el bit 1 es 0
}
```

- ***ROTATE_LEFT(address, bytes)***

Esta función rota a la izquierda un bit de un array o de una estructura. Nótese que la rotación implica que el bit MSB pasa a ser el bit LSB. address puede ser un identificador de un array o la dirección a un byte o a una estructura, por ejemplo, &dato. bytes es el número de bytes implicados en la rotación.

Ejemplo:

```
x = 0x86;
rotate_left( &x, 1);      // x tiene ahora 0x0d
```

- ***ROTATE_RIGHT(address, bytes)***

Esta función rota a la derecha un bit de un array o de una estructura. Nótese que esta rotación implica que el bit LSB pasa a ser el bit MSB. address puede ser un identificador de un array o la dirección a un byte o a una estructura, por ejemplo, &dato. bytes es el número de bytes implicados en la rotación.

Ejemplo:

```
struct {
        int cell_1 : 4;
        nt cell_2 : 4;
        int cell_3 : 4;
        int cell_4 : 4;
        cells;

        rotate_right( &cells, 2);
        rotate_right( &cells, 2);
        rotate_right( &cells, 2);
        rotate_right( &cells, 2);          // celda 1->4, 2->1, 3->2 y 4->3
```

- ***SHIFT_LEFT(address, bytes, value)***

Esta función desplaza a la izquierda un bit de un array o de una estructura. Nótese la diferencia entre rotación y desplazamiento; en la primera se produce una 'realimentación' del dato, en la segunda no.

address puede ser un identificador de array o la dirección de una estructura, por ejemplo, &dato.

bytes es el número de bytes implicados en el desplazamiento.

value es el valor del bit
que insertamos. Esta función devuelve el bit que queda fuera en el desplazamiento.

Ejemplo:

```
byte buffer[3];
for(i=1; i<=24; ++i){
                while (!input(PIN_A2));
                shift_left(buffer,3,input(PIN_A3));
                while (input(PIN_A2)) ;
                }

/* lee 24 bits de la patilla A3; cada bit se lee durante la transición de bajo a
alto en la patilla A2 */
```

- ***SHIFT_RIGHT(address, bytes, value)***

Esta función desplaza a la derecha un bit de un array o de una estructura. Nótese la diferencia entre rotación y desplazamiento; en la primera se produce una 'realimentación' del dato, en la segunda no.

address puede ser un identificador de array o la dirección de una estructura, por ejemplo, &dato.

bytes es el número de bytes implicados en el desplazamiento.

value es el valor del bit que insertamos. Esta función devuelve el bit que queda fuera en el desplazamiento.

Ejemplo:

```
struct { byte time;
        byte command : 4;
        byte source  : 4;
        } msg;

for(i=0; i<=16;++i) {
                while(!input(PIN_A2));
                shift_right(&msg,3,input(PIN_A1));
                while (input(PIN_A2)) ;
                }

/* lee 16 bits de la patilla A1; cada bit se lee en la transición de bajo a alto
en la patilla A2 */

for(i=0;i<8;++i)
                output_bit(PIN_A0, shift_right(&data,1,0));

/ * desplaza 8 bits y los saca por el  Pin_A0; primero sale el LSB * /
```

- ***SWAP(byte)***

Esta función intercambia el nibble alto con el nibble bajo del byte dado. Esto es lo mismo que: byte = (byte << 4) | (byte >> 4);

Ejemplo:

```
x=0x45;
swap(x);          // x ahora tiene  0x54
```

6. DEFINICIÓN DE DATOS

*Si **TYPEDEF*** se pone delante de la definición de un dato, entonces no se asigna espacio de memoria al identificador a menos que sea utilizado como un especificador de tipo en otras definiciones de datos.

Si delante del identificador ponemos ***CONST*** entonces, el identificador es tratado como constante. Las constantes deben ser inicializadas y no pueden cambiar en tiempo de ejecución.

No están permitidos punteros a constantes. ***SHORT*** es un tipo especial utilizado para generar código muy eficiente para las ***operaciones de I/O***.

No se permiten las arrays de ***SHORT*** ni los punteros a ***SHORT***. La siguiente tabla muestra la sintaxis para las definiciones de datos.

Ejemplos:

```
int a,b,c,d;
typedef int byte;
typedef short bit;

bit e,f;
byte g[3][2];
char *h;
enum boolean  {false, true};
boolean j;
byte k = 5;
byte const SEMANAS = 52;
byte const FACTORES [4] = {8, 16, 64, 128};

struct registro_datos {
        byte   a [2];
        byte  b : 2;  /*2 bits */
        byte  c : 3;  /*3  bits*/
        int d;
}
```

DEFINICIONES DE DATOS

typedef	[calificador_tipo] [especificador_tipo] [identificador]
static	Variable global e inicializada a 0
auto	La variable existe mientras el procedimiento está activo Es el valor por defecto, por eso no es necesario poner auto

Especificadores de tipo:

unsigned	define un número de 8 bits sin signo
unsigned int	define un número de 8 bits sin signo
int	define un número de 8 bits sin signo
char	define un número de 8 bits sin signo
long	define un número de 16 bits sin signo
long int	define un número de 16 bits sin signo
signed	define un número de 8 bits con signo
signed int	define un número de 8 bits con signo
signed long	define un número de 16 bits con signo
float	define un número de 32 bits en punto flotante
short	define un bit
short int	define un bit

Identificador	Identificador de una definición TYPE de tipo
Enum	tipo enumerado, véase sintaxis a continuación
Struct	estructura, véase sintaxis a continuación
Unión	unión, véase sintaxis a continuación

declarador:

[const] [*]identificador [expr._constante][= valor_inicial]

enumerador:

enum [identificador]{
[lista_identificadores[= expresión_constante]]

}

estructura y unión:

struct [identificador] {
[calificador_tipo [[*]identificador
:expresión_constante [expresión_constante]]]
}

unión Ídem

Ejemplo:

```
struct lcd_pin_map {
        boolean enable;
        boolean rs;
        boolean rw;
        boolean unused;
        int     data : 4;
    } lcd;
```

Si ponemos *expresión_constante* después de un *identificador*, determina el número de bits que utilizará dicho identificador. Este número puede ser 1,2, ...8. Se pueden utilizar arrays de dimensión múltiple poniendo los [] que se precisen. También podemos anidar estructuras y uniones.

Un identificador después de **struct** puede usarse en otra **struct** y las {} no se pueden usar para reutilizar la estructura de la misma forma otra vez.

Nota: Recordar que todo lo expresado entre [corchetes]es opcional, es decir, se puede especificar o no, según convenga. Si la expresión va entre {llaves} entonces indica que debe aparecer ***una o más veces*** y en ese caso separados por comas; véase más abajo el siguiente ejemplo.

El identificador después de **enum** es un tipo de datos tan grande como la mayor de las constantes de la lista. Cada uno de los identificadores de la lista son creados como una constante. Por defecto, el primer identificador se pone a cero y los siguientes se incrementan en uno. Si incluimos "=*expresión_constante*" después de un identificador éste tendrá el valor de la *expresión_constante* y los siguientes se incrementarán en uno.

Ejemplo:

```
enum ADC_SOURCE {an0, an1, an2, an3, bandgap, srefh, srefl, itemp, refa, refb};
```

7. DEFINICIÓN DE FUNCIÓN

El formato de la definición de una función es como sigue:

[calificador_tipo] identificador ([[especificador_tipo identificador]) {

[cuerpo de la función]
}

El ***calificador_tipo*** para una función pueden ser:

void o un
especificador de tipo (véase la lista de la página anterior)

La definición de una función puede ir precedida por una de las siguientes directivas del proprocesador (***calificadores de función***) para dentificar una característica especial de la función: ***#separate #inline #int_...***

Cuando utilizamos una de las directivas mencionadas y la función tiene un prototipo (declaración anterior a la definición de la función, y colocada al principio del fichero fuente) hay que incluir la misma ***#directiva*** en el *prototipo* y en la *definición de la función*.

Una característica no muy corriente, se ha añadido al compilador para ayudar a evitar los problemas creados por el hecho de que no pueden crearse punteros a constantes de cadenas. Una función que tiene un parámetro de tipo char aceptará una constante de cadena. El compilador generará un bucle que llama a la función una vez para cada carácter de la cadena.

Ejemplo:

```
void lcd_putc(char c ) {
// definición de la función
...
}
lcd_putc ("Máquina parada".);
```

8. FUNCIONES: PARÁMETROS POR REFERENCIA

El compilador está limitado a trabajar con parámetros por referencia.

Esto aumenta la legibilidad del código así como la eficacia de algunos procedimientos.

Los dos procedimientos siguientes son iguales pero sólo el segundo opera con parámetros por referencia:

```
funcion_a(int*x,int*y) {          / * declaración de la función * /

        (*x!=5);
        *y=*x+3;
}

funcion_a(&a,&b);                 /* llamada a la función  */



funct_b(int&x,int&y) {            / * declaración de la función * /

                                  /* paso de parámetros por referencia * /

        if(x!=5)
        y=x+3;
 }

funcion_b(a,b);                   /* llamada a la función  */
```

9. Edición de un programa en C

Para crear un programa hay que seguir los pasos siguientes:

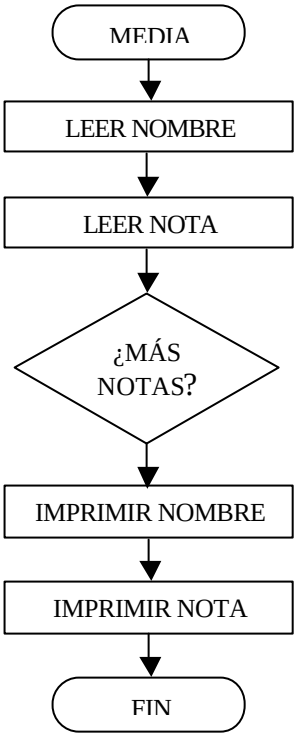
1. Especificaciones del programa (qué tiene que hacer)
2. Hacer organigrama
3. Escribir el código fuente (conocer el lenguaje)
4. Compilar + Enlazar (Link)
5. Depurar errores, si los hay

ANÁLISIS DE UN PROBLEMA SENCILLO

Como ejemplo orientativo, se hace a continuación el desarrollo de un programa sencillo. Se trata de obtener la nota media de un alumno durante un trimestre. El análisis de esta tarea, que la hemos llamado MEDIA, puede dar el siguiente procedimiento:

1. leer NOMBRE
2. leer NOTA
3. si no hay mas notas, ir al punto 5
4. ir al punto 2
5. calcular la MEDIA
6. imprimir NOMBRE
7. imprimir MEDIA

DIAGRAMA DE FLUJO DEL EJEMPLO "MEDIA"



10. Estructura de un programa en C

De forma generalizada, la estructura de un programa en C tiene el siguiente aspecto:

```
declaraciones globales
prototipos de funciones

main() {

variables locales;
bloque de sentencias;
llamadas a las funciones;

}

funcion_1() {

variables locales a funcion_1;
bloque de sentencias;
llamada a otra/s funciones;

}


funcion_n() {

...

}
```

EJEMPLO DE UN PROGRAMA EN LANGUAJE C

```
/* parpadeo.c Programa que hace parpadear un led en RB7 cada ms */

#include <16C84.H>                  /* tipo de PIC */

#use delay( clock = 4000000 )       /* reloj de 4 MHz */
#byte      puerto_b = 06            /* dirección del puerto B  */

void main( void )
{
 set_tris_b( 0x00 );                /*  puerto B como salida */
 puerto_b = 0;                      /* apaga todos los led */

do{

delay_us( 1000 );           /* retardo de 1000 (seg. */
bit_set( puerto_b, 7 );     /* enciende el led  RB7 */
delay_us( 1000 );           /* espera  1 ms*/
bit_clear( puerto_b, 7);    /* apaga el led */
} while( TRUE );                    /* Repetir siempre */
}
```

11. MENSAJES DE ERROR DEL COMPILADOR

#ENDIF with no corresponding #IF
A numeric expression must appear here. The indicated item must evaluate to a number.

A #DEVICE required before this line
The compiler requires a #device before it encounters any statement or compiler directive that may cause it to generate code. In general #defines may appear before a #device but not much more.

A numeric expression must appear here
Some C expression (like 123, A or B+C) must appear at this spot in the code. Some expression that will evaluate to a value.

Array dimensions must be specified
The [] notation is not permitted in the compiler. Specific dimensions must be used. For example A[5].

Arrays of bits are not permitted
Arrays may not be of SHORT INT. Arrays of Records are permitted but the record size is always rounded up to the next byte boundary.

Attempt to create a pointer to a constant
Constant tables are implemented as functions. Pointers cannot be created to functions. For example CHAR CONST MSG[9]={"HI THERE"}; is permitted, however you cannot use &MSG. You can only reference MSG with subscripts such as MSG[i] and in some function calls such as Printf and STRCPY.

Attributes used may only be applied to a function (INLINE or SEPARATE)
An attempt was made to apply #INLINE or #SEPARATE to something other than a function.

Bad expression syntax
This is a generic error message. It covers all incorrect syntax.

Baud rate out of range
The compiler could not create code for the specified baud rate. If the internal UART is being used the combination of the clock and the UART capabilities could not get a baud rate within 3% of the requested value. If the built in UART is not being used then the clock will not permit the indicated baud rate. For fast baud rates, a faster clock will be required.

BIT variable not permitted here
Addresses cannot be created to bits. For example &X is not permitted if X is a SHORT INT.

Can.t change device type this far into the code
The #DEVICE is not permitted after code is generated that is device specific. Move the #DEVICE to an area before code is generated.

Character constant constructed incorrectly
Generally this is due to too many characters within the single quotes. For example 'ab' is an error as is '\nr'. The backslash is permitted provided the result is a single character such as '\010' or '\n'.

Constant out of the valid range
This will usually occur in inline assembly where a constant must be within a particular range and it is not. For example BTFSC 3,9 would cause this error since the second operand must be from 0-8.

Constant too large, must be < 65536
As it says the constant is too big.

Define expansion is too large
A fully expanded DEFINE must be less than 255 characters. Check to be sure the DEFINE is not recursively defined.

Define syntax error
This is usually caused by a missing or mis-placed (or) within a define.

Different levels of indirection
This is caused by a INLINE function with a reference parameter being called with a parameter that is not a variable. Usually calling with a constant causes this.

Divide by zero
An attempt was made to divide by zero at compile time using constants.

Duplicate case value
Two cases in a switch statement have the same value.

Duplicate DEFAULT statements
The DEFAULT statement within a SWITCH may only appear once in each SWITCH. This error indicates a second DEFAULT was encountered.

Duplicate #define
The identifier in the #define has already been used in a previous #define. The redefine an identifier use #UNDEF first. To prevent defines that may be included from multiple source do something like:

- #ifndef ID
- #define ID text
- #endif

Duplicate function
A function has already been defined with this name. Remember that the compiler is not case sensitive unless a #CASE is used.

Duplicate Interrupt Procedure
Only one function may be attached to each interrupt level. For example the #INT_RB may only appear once in each program.

Duplicate USE
Some USE libraries may only be invoked once since they apply to the entire program such as #USE DELAY. These may not be changed throughout the program.

Element is not a member
A field of a record identified by the compiler is not actually in the record. Check the identifier spelling.

ELSE with no corresponding IF
Check that the {and} match up correctly.

End of file while within define definition
The end of the source file was encountered while still expanding a define. Check for a missing).

End of source file reached without closing comment */ symbol
The end of the source file has been reached and a comment (started with /*) is still in effect. The */ is missing.

Error in define syntax

Error text not in file
The error is a new error not in the error file on your disk. Check to be sure that the errors.txt file you are using came on the same disk as the version of software you are executing. Call CCS with the error number if this does not solve the problem.

Expect ;
Expect comma
Expect WHILE
Expect }
Expecting :
Expecting =
Expecting a (
Expecting a , or)
Expecting a , or }
Expecting a .
Expecting a ; or ,
Expecting a ; or {
Expecting a close paren
Expecting a declaration
Expecting a structure/union
Expecting a variable
Expecting a]
Expecting a {
Expecting an =
Expecting an array
Expecting an expression
Expecting an identifier
Expecting an opcode mnemonic
This must be a Microchip mnemonic such as MOVLW or BTFSC.

Expecting LVALUE such as a variable name or * expression
This error will occur when a constant is used where a variable should be. For example 4=5; will give this error.

Expecting a basic type
Examples of a basic type are INT and CHAR.

Expecting procedure name
Expression must be a constant or simple variable
The indicated expression must evaluate to a constant at compile time. For example 5*3+1 is permitted but 5*x+1 where X is a INT is not permitted. If X were a DEFINE that had a constant value then it is permitted.

Expression must evaluate to a constant
The indicated expression must evaluate to a constant at compile time. For example 5*3+1 is permitted but 5*x+1 where X is a INT is not permitted. If X were a DEFINE that had a constant value then it is permitted.

Expression too complex
This expression has generated too much code for the compiler to handle for a single expression. This is very rare but if it happens, break the expression up into smaller parts.

Too many assembly lines are being generated for a single C statement. Contact CCS to increase the internal limits.

Extra characters on preprocessor command line
Characters are appearing after a preprocessor directive that do not apply to that directive. Preprocessor commands own the entire line unlike the normal C syntax. For example the following is an error:

#PRAGMA DEVICE <PIC16C74> main() { int x; x=1;}

File in #INCLUDE can not be opened
Check the filename and the current path. The file could not be opened.

Filename must start with " or <

Filename must terminate with " or >

Floating-point numbers not supported
A floating-point number is not permitted in the operation near the error. For example, ++F where F is a float is not allowed.

Function definition different from previous definition
This is a mis-match between a function prototype and a function definition. Be sure that if a #INLINE or #SEPARATE are used that they appear for both the prototype and definition. These directives are treated much like a type specifier.

Function used but not defined
The indicated function had a prototype but was never defined in the program.

Identifier is already used in this scope
An attempt was made to define a new identifier that has already been defined.

Illegal C character in input file
A bad character is in the source file. Try deleting the line and re-typing it.

Improper use of a function identifier
Function identifiers may only be used to call a function. An attempt was made to otherwise reference a function. A function identifier should have a (after it.

Incorrectly constructed label
This may be an improperly terminated expression followed by a label. For example:
x=5+
MPLAB:

Initialization of unions is not permitted
Structures can be initialized with an initial value but UNIONS cannot be.

Internal compiler limit reached
The program is using too much of something. An internal compiler limit was reached. Contact CCS and the limit may be able to be expanded.

Invalid conversion from LONG INT to INT
In this case, a LONG INT cannot be converted to an INT. You can type cast the LONG INT to perform a truncation. For example:
I = INT(LI);

Internal Error - Contact CCS
This error indicates the compiler detected an internal inconsistency. This is not an error with the source code; although, something in the source code has triggered the internal error. This problem can usually be quickly corrected by sending the source files to CCS so the problem can be re-created and corrected.

In the meantime if the error was on a particular line, look for another way to perform the same operation. The error was probably caused by the syntax of the identified statement. If the error was the last line of the code, the problem was in linking. Look at the call tree for something out of the ordinary.

Invalid parameters to shift function

Built-in shift and rotate functions (such as SHIFT_LEFT) require an expression that evaluates to a constant to specify the number of bytes.

Invalid ORG range
The end address must be greater than or equal to the start address. The range may not overlap another range. The range may not include locations 0-3. If only one address is specified it must match the start address of a previous #org.

Invalid Pre-Processor directive
The compiler does not know the preprocessor directive. This is the identifier in one of the following two places:
#xxxxx
#PRAGMA xxxxx

Library in USE not found
The identifier after the USE is not one of the pre-defined libraries for the compiler. Check the spelling.

LVALUE required
This error will occur when a constant is used where a variable should be. For example 4=5; will give this error.

Macro identifier requires parameters
A #DEFINE identifier is being used but no parameters were specified ,as required. For example:
#define min(x,y) ((x<y)?x:y)
When called MIN must have a (--,--) after it such as:
r=min(value, 6);

Missing #ENDIF
A #IF was found without a corresponding #ENDIF.

Missing or invalid .REG file
The user registration file(s) are not part of the download software. In order for the software to run the files must be in the same directory as the .EXE files. These files are on the original diskette, CD ROM or e-mail in a non-compressed format. You need only copy them to the .EXE directory. There is one .REG file for each compiler (PCB.REG, PCM.REG and PCH.REG).

Must have a #USE DELAY before a #USE RS232
The RS232 library uses the DELAY library. You must have a #USE DELAY before you can do a #USE RS232.

No MAIN() function found
All programs are required to have one function with the name main().

Not enough RAM for all variables
The program requires more RAM than is available. The memory map (ALT-M) will show variables allocated. The ALT-T will show the RAM used by each function. Additional RAM usage can be obtained by breaking larger functions into smaller ones and splitting the RAM between them.

For example, a function A may perform a series of operations and have 20 local variables declared. Upon analysis, it may be determined that there are two main parts to the calculations and many variables are not shared between the parts. A function B may be defined with 7 local variables and a function C may be defined with 7 local variables. Function A now calls B and C and combines the results and now may only need 6 variables. The savings are accomplished because B and C are not executing at the same time and the same real memory locations will be used for their 6 variables (just not at the same time). The compiler will allocate only 13 locations for the group of functions A, B, C where 20 were required before to perform the same operation.

Number of bits is out of range
For a count of bits, such as in a structure definition, this must be 1-8. For a bit number specification, such as in the #BIT, the number must be 0-7.

Out of ROM, A segment or the program is too large
A function and all of the INLINE functions it calls must fit into one segment (a hardware code page). For example, on the '56 chip a code page is 512 instructions. If a program has only one function and that function is 600 instructions long, you will get this error even though the chip has plenty of ROM left. The function needs to be split into at least two smaller functions. Even after this is done, this error may occur since the new function may be only called once and the linker might automatically INLINE it. This is easily determined by reviewing the call tree via ALT-T. If this error is caused by too many functions being automatically INLINED by the linker, simply add a #SEPARATE before a function to force the function to be SEPARATE. Separate functions can be allocated on any page that has room. The best way to understand the cause of this error is to review the calling tree via ALT-T.

Parameters not permitted
An identifier that is not a function or preprocessor macro can not have a (after it.

Pointers to bits are not permitted
Addresses cannot be created to bits. For example, &X is not permitted if X is a SHORT INT.

Pointers to functions are not valid
Addresses cannot be created to functions.

Previous identifier must be a pointer
A -> may only be used after a pointer to a structure. It cannot be used on a structure itself or other kind of variable.

Printf format type is invalid
An unknown character is after the % in a printf. Check the printf reference for valid formats.

Printf format (%) invalid
A bad format combination was used. For example, %lc.

Printf variable count (%) does not match actual count
The number of % format indicators in the printf does not match the actual number of variables that follow. Remember in order to print a single %, you must use %%.

Recursion not permitted
The linker will not allow recursive function calls. A function may not call itself and it may not call any other function that will eventually re-call it.

Recursively defined structures not permitted
A structure may not contain an instance of itself.

Reference arrays are not permitted
A reference parameter may not refer to an array.

Return not allowed in void function
A return statement may not have a value if the function is void.

String too long

Structure field name required
A structure is being used in a place where a field of the structure must appear. Change to the form s.f where s is the structure name and f is a field name.

Structures and UNIONS cannot be parameters (use * or &)

A structure may not be passed by value. Pass a pointer to the structure using &.

Subscript out of range
A subscript to a RAM array must be at least 1 and not more than 128 elements. Note that large arrays might not fit in a bank. ROM arrays may not occupy more than 256 locations.

This expression cannot evaluate to a number
A numeric result is required here and the expression used will not evaluate to a number.

This type cannot be qualified with this qualifier
Check the qualifiers. Be sure to look on previous lines. An example of this error is:
VOID X;

Too many #DEFINE statements
The internal compiler limit for the permitted number of defines has been reached. Call CCS to find out if this can be increased.

Too many array subscripts
Arrays are limited to 5 dimensions.

Too many constant structures to fit into available space
Available space depends on the chip. Some chips only allow constant structures in certain places. Look at the last calling tree to evaluate space usage. Constant structures will appear as functions with a @CONST at the beginning of the name.

Too many identifiers have been defined
The internal compiler limit for the permitted number of variables has been reached. Call CCS to find out if this can be increased.

Too many identifiers in program
The internal compiler limit for the permitted number of identifiers has been reached. Call CCS to find out if this can be increased.

Too many nested #INCLUDEs
No more than 10 include files may be open at a time.

Too many parameters
More parameters have been given to a function than the function was defined with.

Too many subscripts
More subscripts have been given to an array than the array was defined with.

Type is not defined
The specified type is used but not defined in the program. Check the spelling.

Type specification not valid for a function
This function has a type specifier that is not meaningful to a function.

Undefined identifier
The specified identifier is being used but has never been defined. Check the spelling.

Undefined label that was used in a GOTO
There was a GOTO LABEL but LABEL was never encountered within the required scope. A GOTO cannot jump outside a function.

Unknown device type
A #DEVICE contained an unknown device. The center letters of a device are always C regardless of the actual part in use. For example, use PIC16C74 not PIC16RC74. Be sure the correct compiler is being used for the indicated device. See #DEVICE for more information.

Unknown keyword in #FUSES
Check the keyword spelling against the description under #FUSES.

Unknown type
The specified type is used but not defined in the program. Check the spelling.

USE parameter invalid
One of the parameters to a USE library is not valid for the current environment.

USE parameter value is out of range
One of the values for a parameter to the USE library is not valid for the current environment.