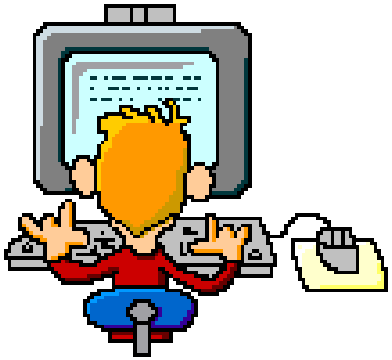


Programación en C

Programación en Lenguaje C

1.- Fundamentos de C

2.- Variables

3.- Operadores

4.- Punteros, arrays y strings

5.- Estructuras y uniones

6.- Funciones

7.- Instrucciones de control de programa

8.- C específico de los PIC:

- Directivas de pre-procesado
- Funciones integradas en el compilador

1.- Los Fundamentos del lenguaje C

• Directivas de Preprocesado: son comandos para el preprocesador de C que indican al compilador cómo debe realizar la generación del código máquina

• Declaraciones: indican los nombres y los atributos asignados a las variables, funciones y tipos que se van a utilizar en el programa

• Definiciones: establecen los contenidos que se van a almacenar en las variables y también qué es lo que van a generar las funciones

• Expresiones: combinación de operadores y operandos que porporcionan un valor final único

• Sentencias de control: establecen la secuencia y el orden de ejecución del programa

• Comentarios: imprescindibles como documentación y explicación del código fuente

• Funciones: conjunto de declaraciones, definiciones, expresiones e instrucciones que desarrollan una tarea específica. Las llaves: { } encierran el cuerpo de las funciones. Las funciones en C no pueden "anidarse" en cuanto a definición

• Función Principal (*Main Function*): todos los programas en C deben contener una función llamada ***main( )*** donde se inicia la ejecución del programa. Las llaves { } que enmarcan el cuerpo de esta función definen el inicio y el final del programa.

Componentes y sintaxis del código fuente :

Los elementos fundamentales en C son las Sentencias y las Funciones.

Las sentencias son las que realmente realizan las operaciones.

Todos los programas en C tienen una o varias funciones. Éstas son subrutinas que contienen una o más sentencias y que pueden ser llamadas por otras partes del programa

Las sentencias dentro de una función se ejecutan secuencialmente empezando por el carácter de llave de apertura { y finalizando con la llave de cierre }

Las llaves también marcan el inicio y el final de bloques de código

El final de las sentencias se marca con el carácter de punto y coma (;).

El carácter de fin de línea no es reconocido por el C como fin de sentencia, por tanto no hay restricciones en cuanto a la posición de las sentencias en la línea ni en el número de sentencias que se pueden situar en una misma línea de código o el número de líneas que puede ocupar una sentencia.

```
suma=M_CABEZA
     +
     dir
     +comandos
     +datos
     ;
```

Como ejemplo:

Estas dos sentencias de asignación son igual de válidas y son equivalentes pero está claro que es más fácil leer e interpretar la segunda que la primera

```
suma=M_CABEZA+dir+comandos+datos;
```

El código en C puede convertirse en críptico y difícil de interpretar y leer ya que permite una gran flexibilidad a la hora de realizar la escritura del código

Cuando se escribe el código, el empleo de las tabulaciones, líneas en blanco y comentarios mejorará la legibilidad del código para uno mismo (al cabo del tiempo puede ser necesario modificarlo) y para los demás en el supuesto de que se deba realizar una transferencia de información

Sintaxis de los Comentarios

Los comentarios se incluyen en el código fuente para explicar el sentido y la intención del código al que acompañan. Son ignorados por el compilador y no afectan a la longitud ni rapidez de ejecución del código final.

Un comentario se puede colocar en cualquier lugar del programa y pueden tener la longitud y el número de líneas que uno quiera

Los comentarios tienen dos formatos posibles

a) Empiezan por /* y finalizan con */, en este caso no pueden anidarse

```
/* Esto es un comentario */

/* Pero este comentario /* parece */ pero no es válido */
```

b) Empiezan por // y finalizan con el final de la línea

```
// Esto también es un comentario válido
```

2.- Variables

Una variable es un nombre asignado a una o varias posiciones de memoria RAM

En C es necesario declarar todas las variables antes de poder utilizarlas, en la declaración se indica el nombre asignado y el tipo de datos que en ella se van a almacenar (opcionalmente también el valor inicial asignado)

La manera en que se almacenan los datos es un aspecto importante en C y más si se tiene en cuenta las limitaciones propias de un microcontrolador

Las declaraciones de variables son sentencias y por tanto deben terminar con ;
la sintaxis de declaración más simple es la siguiente:

```
tipo nombre_variable;
```

p.e.: int i;

tipo — es uno de los tipos de datos válidos en C
nombre_variable — es el nombre que le asignamos

(En PCM: un identificador puede tener hasta 32 caracteres empezando siempre por letra)

Las variables pueden declararse dentro de una función (variables locales) o fuera de todas las funciones (variables globales)

Las variables locales pueden usarse sólo en sentencias presentes dentro de la función en la que fueron declaradas. Las variables locales se crean cuando se "entra" en la función y se destruyen cuando se sale

Las variables locales se deben declarar al principio de la función y antes de las sentencias.

Es válido que variables locales en diferentes funciones tengan el mismo nombre

Las variables globales se pueden utilizar por parte de todas las funciones y deben declararse antes de cualquier función que las use

Además del tipo de dato que van a almacenar, cada variable puede tener especificado otro atributo más que es la clase de almacenamiento que puede ser automática (auto), externa (extern), estática (static) y registro (register)

Clase de Almacenamiento de una Variable

Las variables necesitan dos atributos en C: el tipo y la clase de almacenamiento

Las clases de almacenamiento posibles para los compiladores de CCS (PCM) son dos:

auto — Es la clase por defecto. Cuando se entra en un bloque de código, el compilador asigna espacio de RAM a las variables declaradas y libera esas posiciones cuando se salga de esa zona, esas mismas posiciones de memoria pueden y serán usadas por otros bloques de código

static — Las variables con esta clase de almacenamiento, son variables permanentes que retendrán los valores que tenían en el momento en que se salió del bloque anteriormente. Se diferencian de las variables globales en que no son conocidas fuera de su función pero mantienen sus valores entre llamadas

extern y register se reconocen como identificadores pero sin efecto en PCM

Tipos de Datos

El lenguaje C estándar (ANSI) admite 5 tipos de datos básicos:

char	(carácter),
int	(entero),
float	(coma flotante en 32 bits),
double	(coma flotante en 64 bits) y
void	(no devuelve ningún valor)

el resto de los tipos de datos se basan en alguno de los anteriores, definiendo los nuevos tipos mediante modificadores que se añaden a los tipos básicos

En coma flotante los bits se dividen en dos campos: Mantisa y Exponente de modo que

$$\text{número} = \text{Mantisa (23 bits)} * 2^{(\text{exponente(8 bits)}-127)}$$

se almacenan en 32 bits:

S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM

En el caso del compilador PCM se tienen los siguientes tipos especificados:

Los tipos básicos en C

Especificación	Significado	Tamaño	Rango
char	carácter	8 bits	0 a 255 (sin signo)
int	entero	8 bits	0 a 255 (sin signo)
float	coma flotante	32 bits	6 bits de precisión
double	float doble precisión	no soportado	No para PCM
void	sin valor	nulo	ninguno
int1	entero de 1 bit	1 bit	0 a 1
int8	entero de 8 bits	8 bits	0 a 255 (sin signo)
int16	entero de 16 bits	16 bits	0 a 65535 (sin signo)
int32	entero de 32 bits	32 bits	0 a $(2^{32}-1)$
short	entero de 1 bit	1 bit	0 a 1
long	entero de 16 bits	16 bits	0 a 65535 (sin signo)

short y long pueden tener detrás la palabra int sin efecto alguno

Todos los tipos de datos son por defecto sin signo (unsigned) salvo float

Si se desea almacenar datos con signo, se debe introducir el modificador signed delante del tipo, el efecto que tendría sería la modificación del rango, pasando a emplearse el convenio de representación en complemento a 2 para los números negativos

Especificación	Significado	Tamaño	Rango
signed char	carácter con signo	8 bits	-128 a 127
signed int	entero con signo	8 bits	-128 a 127
signed long	coma flotante	16 bits	-32768 a 32767

El C permite operar con distintos tipos de datos en una misma expresión. En ese caso, se aplican una serie de reglas de conversión para resolver las diferencias.

Se produce una "promoción" hacia los tipos de datos de mayor longitud que estén presentes en la expresión.

Tipo de datos Enumerados

En C resulta posible definir una lista de constantes enteras como valores que puede tomar una variable. Esta lista de constantes creada con una enumeración pueden colocarse en cualquier lugar donde se pueda situar un entero.
Declaración de Enumeración:

```
enum nombre_tipo {lista enumerada}  [variable(s)];
```

p.e.:
```
enum colores {rojo, azul, verde, amarillo} micolor;
enum comidas {desayuno, almuerzo, merienda, cena};
enum boolean {true, false}
```

Una vez que se ha definido la enumeración para una variable, puede usarse para definir otras variables en otros puntos del programa

p.e.:
```
colores otro_color;  //declaración variable otro_color de tipo colores
```

El compilador asigna valores enteros consecutivos a cada elemento de la lista empezando por el cero (0):

```
micolor=verde+azul; //el resultado es el amarillo
```

Sentencia typedef

Esta sentencia se emplea para dar un nombre nuevo a un tipo de datos ya existente. A partir de ahí el nuevo tipo se puede utilizar en las declaraciones de variables. El formato es

```
typedef nombre_antiguo nombre_nuevo;
```

p.e.: typedef int byte; typedef short bit;

bit e,f; byte g;

La definición de un nombre nuevo para un tipo de dato no desactiva el uso del antiguo que sigue siendo válido. Se pueden utilizar muchas sentencias typedef para el mismo tipo original.

El empleo de typedef no genera código. Es una sentencia que contribuye a la portabilidad y a la legibilidad del código

Asignación de Variables

La asignación es una sentencia con el formato:

nombre_variable = expresión;

p.e.: i=0; i=j;

la expresión más simple sería un valor constante. Existen muchos valores constantes posibles en C: un elemento de una lista enumerada, un número, un carácter,.... Una constante de tipo carácter va encerrada entre comillas simples ('m').

Cuando se declara una variable también es posible asignarle un valor inicial

p.e.: char letra = 'A'; int cuanto=100;

las variables globales con asignación inicial sólo se inicializan al principio del programa, las variables locales se inicializan cada vez que se entra en el bloque en el que han sido declaradas

Constantes

Son valores fijos que no pueden modificarse por programa. Pueden ser cualquier tipo de dato básico. Cuando el compilador encuentra una constante determina de qué tipo es en función de su escritura y la ajustará al menor tipo de datos compatible

Las variables de tipo const no pueden cambiarse durante la ejecución de un programa pero sí que se le puede asignar un valor inicial que se mantendrá a lo largo del programa:

p.e. `const int a=10;`

También se podrían declarar constantes con la directiva #define:

```
#define  nombre  valor
```

p.e.
```
#define pi 3.141516
#define guion '-'
```

Constantes (II): su representación para PCM

Decimales:	112
Octales:	0112
Hexadecimales:	0x112
Binarias:	0b11001001
Carácter:	'A'
Código de carácter en octal:	'\010'
Código de carácter en hexadecimal:	'\xA5'
Caracteres especiales ('\c'):	'\n' avance de línea (como '\x0a'), '\r' retorno de carro (= '\x0d'),…
Tira de caracteres:	"ABCDEF" **(con carácter nulo al final)**

3.- Operadores en C

El lenguaje C define más operadores que la mayor parte de los otros lenguajes de programación. Una expresión será una combinación de operadores y de operandos (que serán variables y/o constantes)

Abreviaturas en C: todos los operadores que requieren de dos operandos admiten abreviaturas de manera que:

variable = variable operador expresión

es equivalente a escribir

variable operador = expresión

Las abreviaturas son muy habituales en los códigos en C "profesionales"

Ejemplos:

a=a+b	equivale a	a+=b
a=a-b	equivale a	a-=b
a=a*b	equivale a	a*=b
a=a/b	equivale a	a/=b

• Operadores Aritméticos: (pueden usarse con cualquier tipo de datos salvo %)

+	Suma	c=a+b;
-	Resta y cambio de signo	c=a-b; c=-a;
*	Multiplicación	c=a*b;
/	División	c=a/b;
%	Módulo (es el resto de una división entera)	c=a%b; sólo con enteros

• Operadores Incremento y Decremento: son operadores que suman 1 y restan 1

++	Incremento	x++ ó ++x equivalen a x=x+1
--	Decremento	x-- ó --x equivalen a x=x-1

Cuando el operador ++ ó -- precede a la variable, (++x,--x) en una expresión, primero se incrementa (o decrementa) la variable y luego se evalúa la expresión. Sin embargo, si ++ ó -- va después de la variable, primero se evalúa la expresión con el valor inicial y luego se incrementa o decrementa.

```
p.e.:   x=14;   z=++x;   //z tomará ahora el valor 15 y x también
        a=12;   z=a++    //z tomará el valor 12 y a el valor 13
```

• Operadores Relacionales: comparan dos valores y devuelven cierto (true) ó falso (false) como resultado de la comparación.

>	Mayor que	a>b
>=	Mayor o igual que	a>=b
<	Menor que	a<b
<=	Menor o igual que	a<=b
==	Igual que	a==b
!=	No igual que	a!=b

el resultado en todos los casos será siempre 0 ó 1, en C se considera cierto todo valor distinto de 0 y falso está definido siempre como 0

• Operadores Lógicos: también devuelven 0 si resultado falso y 1 si verdadero

&&	Función lógica AND
||	Función lógica OR
!	Función lógica NOT

los op. lógicos y los relacionales se utilizan conjuntamente en los condicionales

p.e.: (cont>max) || ((max==57) && (cont>=10))

• Operadores a nivel de bits: estos operadores permiten operaciones bit a bit entre datos de tipo entero o carácter o sus variantes (exclusivamente), el resultado es también un byte considerado como entero o carácter

&	AND bit a bit	a & b
|	OR bit a bit	a | b
^	XOR bit a bit	a ^ b
~	Complemento a 1	~a
>>	Desplazamiento a la dcha. n veces	a >> n
<<	Desplazamiento a la izq. n veces	a << n

```
Ejemplo:        a = 0b11010011;
                b= 0b01010001;
c=a&b;          //c será 0b01010001
c=a|b;          //c será 0b11010011
c=a^b;          //c toma el valor 0b10000010
c= ~a;          //c adopta el valor 0b00101100
c= a >> 3;      //c: 0b00011010 entran 3 ceros por izq. y "pierde" los 3 salientes
c= b << 3;      //c: 0b10001000 entran 3 ceros por dcha. y "pierde" los 3 salientes
```

Precedencia de Operadores

La precedencia de los operadores indica el orden en el que éstos son procesados por el compilador de C cuando aparecen varios en una misma expresión. Los paréntesis se usarán para marcar el orden específico deseado

Prioridad	Operador	Ejemplo
1	( ) Paréntesis	(a+b)/c
2	!, ~, ++, --, -, * (cont.), & (dir.), sizeof(*)	a=&b+c
3	*, /, %	a%b
4	+, -	b-a
5	<<, >>	a=b>>c
6	<, <=, >, >=	a>=b
7	==, !=, & (AND bit a bit)	a=b&c
8	| (OR bit a bit)	x=y|z
9	^ (XOR bit a bit)	i=j^k
10	&& (AND lógico)	a&&b
11	|| (OR lógico)	a||b
12	=, +=, -=, *=, /=	a=b

(*) Devuelve la longitud en bytes de la variable o tipo al que precede

Ejemplos de Precedencia de Operadores:

```
        10-2*5          se evalúa como  0
        (10-2)*5        toma el valor de 40
        cont*num+80/val-39%cont
equivale a:
        (cont*num)+(80/val)-(39%cont)
```

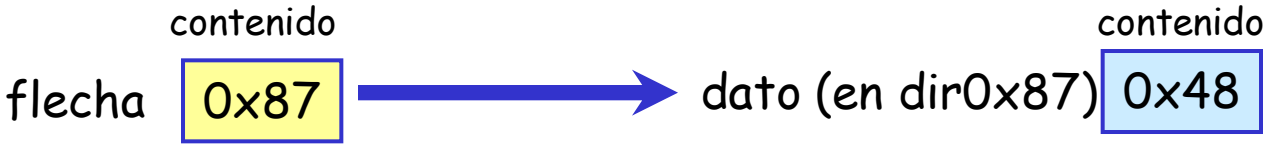
Ejemplo

```
        int a=0,b=0;

        a=6*8+3*b++; //tras esta expresión a=48 y b=1

        b+= -- a*2 + 3*4;

/*      1º  se decrementa a: a=47
        2º se hacen los productos a*2 (=94) y 3*4 (=12)
        3º se suman los productos anteriores (=106)
        4º se suma el valor anterior a b y se le asigna a b (=107 finalmente) */
```

4.- Punteros y Arrays

• Un puntero es una posición de memoria (variable) que almacena la dirección de otra posición de memoria

p.e.: si la variable puntero llamada flecha contiene la dirección de una variable llamada dato, entonces se dice que flecha apunta a dato



• Para declarar una variable de tipo puntero se utiliza la sintáxis:

```
tipo *nombre_variable;
```

tipo indica el tipo de variable a la que nombre_variable puede apuntar.
El asterisco anterior al nombre no forma parte del mismo sino que es una indicación al compilador de que nombre_variable es un puntero

p.e.: int *apunta; //apunta es una variable puntero a un entero

- Hay dos operadores especiales asociados con los punteros (& y *):

  &var devuelve la dirección donde se almacena la variable var

  *ptr devuelve el contenido de la dirección apuntada por el puntero ptr

p.e.:

```
main( )
{
        int     *ap,b;   //se declara ap como puntero a enteros y b
                         //como una variable entero
        b=6;     //Asignamos 6 a la variable b
        ap=&b;   //asignamos la dirección de b a a
        c=*ap;   //c toma ahora el valor 6
        *ap=10; //ahora el contenido de la dirección apuntada por ap
                 //pasa a ser 10, como ap sigue apuntando a b
                 //entonces b pasa a tomar el valor 10
}
```

Otro ejemplo:

dirección		nombre variable
0x70	0xA3	i
0x71	0x67	j
0x72	0x34	k
0x73	0x72	ptr

Declaración de variables

```
int i,j,k;
int *ptr;
```

con los valores arriba indicados se tiene:

i	es	0xA3
&i	es	0x70
ptr	es	0x72
*ptr	es	0x34

Restricciones de los Punteros

Los punteros se pueden tratar como otras variables genéricas aunque con ciertas reglas y excepciones:

- Sólo se pueden aplicar 4 operadores a los punteros: +, ++, - y --
- Sólo se pueden añadir o restar valores enteros
- Cuando se incrementa un puntero (p++), pasa a apuntar a la siguiente posición de memoria y se incrementa en la cantidad de bytes que ocupa el tipo de variable a la que apunta

p.e.
```
int16    *ptr, num; //ptr apunta a enteros de 16 bits
int      *apunta, cuanto; //apunta señala a enteros 8 bits
ptr=100;  apunta=200;
ptr++; //pasa a tomar el valor 102
apunta++; //adopta el valor 201
```

- El orden de precedencia del operador * también es importante:
```
*p++;    //-> primero incrementa y luego nos da el dato apuntado
(*p)++; //-> incrementa el dato apuntado por p
```

Arrays (¿Arreglos o Matrices?)

Un array es una lista de variables del mismo tipo que pueden ser referenciadas con un mismo nombre. Cada variable individual se denomina elemento del array. Es una manera simple de manejar grupos de datos relacionados.

La declaración de un array unidimensional (vector) tiene la siguiente sintáxis:

```
tipo nombre_variable [tamaño del array]
```

donde tipo es un tipo de datos válido en C, nombre_variable es el nombre del array y tamaño del array indica el número de elementos que hay en el array.

p.e.: `int num[10]; //declara un array de 10 enteros`

cada elemento de un array se identifica por un índice, el primer elemento tiene asignado el índice 0 (num[0]) y el último índice tamaño-1 (num[9])

Los elementos de un array unidimensional se almacenan en posiciones de memoria contiguas, ocupando el primer elemento (el de índice 0) la posición más baja.

Un elemento de un array puede sustituir en una expresión a cualquier variable o constante del tipo especificado. C no permite asignar el valor de un array a otro array con una simple asignación:

p.e.: int a[10], b[10];

NO ~~a=b~~; //se debe realizar una asignación elemento a elemento

También es posible crear arrays de varias dimensiones, en la definición:

tipo nombre_variable [num.filas][num.columnas][…]

y se hace referencia a los elementos con varios paréntesis:

p.e. int matriz[3][4]; //definimos matriz de enteros de 3 x 4
matriz[0][1]=12; //asignación a uno de los elementos

Strings (Tiras de Caracteres o Arrays unidimensionales de caracteres)

Una tira de caracteres (o string) en C se trata como un array de caracteres de dimensión uno que se trata de manera conjunta y finaliza con un carácter nulo (un 0 -cero-). Existen muchas funciones para manipular strings en C.

Cuando se declara una tira de caracteres se debe añadir un elemento más, el carácter nulo (que se especifica \0) al tamaño máximo de caracteres esperado

p.e. char tira_caracteres[11]; //declaración para string

Las arrays de strings son muy comunes en C y se declaran como cualquier array multidimensional

p.e. char nombres[10,41]; //para almacenar hasta 10 nombres de 40 car.

si se quiere luego acceder a un nombre de dicho array se puede especificar sólo el primero de los índices:

printf("%s", nombres[5]); //Realiza la impresión del nombre[5]

Inicialización de Arrays

Es posible realizar una asignación inicial a los elementos de un array con una lista de valores:

```
tipo  nombre_array [tamaño] = (lista de valores);
```

la lista de valores es una lista de constantes compatibles con el tipo declarado y separadas por comas. La asignación se hace por orden

p.e.
```
int numero[5] = (23, 34, 0, 12, 5);
                //numero[0]=23, numero[1]=34, numero[2]=0,…
```

Una tira de caracteres (string) puede inicializarse carácter a carácter o de manera conjunta, en cuyo caso el carácter nulo lo añade el compilador:

p.e.
```
char str[4] = ('H', 'O', 'L', 'A');  //No hay carácter nulo al final
char otra_str[5] = "HOLA"; // Se añade car. nulo al final
```

Punteros y Arrays

En C punteros y arrays están íntimamente relacionados, hasta el punto de ser intercambiables en ocasiones. Cuando se usa un nombre de array sin índice, lo que estamos haciendo realmente es utilizar un puntero a la posición del primer elemento del array.

Cuando se pasa un array a una función, lo que se está pasando es un puntero al primero de los elementos. Puesto que un array sin índice es un puntero, se puede asignar ese valor a otro puntero

p.e.
```
int vector[4] = (8, 23, 45, 12);
main( )
{       int *p,i,j;                    //Se declara un puntero a enteros
        p=vector; //Ahora p apunta al primer elemento de vector
        i=*(p+2);  //es válida la asignación i toma el valor 45
        j=p[3];    //también es válida y j toma el valor 12
}
```

5.- Estructuras y Uniones

En lenguaje C una estructura es una colección de variables que se referencian mediante un único nombre. Cada variable de una estructura puede ser de diferentes tipos de datos. La sintaxis de definición es la siguiente:

```
struct especificador_tipo_estructura
{
        tipo campo_1;
        tipo campo_2;
        ...
        tipo campo_n;
} lista_variables tipo_esta_estructura;
```

Nombre de la estructura definida
(si sólo se va a definir una var. se puede prescindir del nombre)

Variables con la estructura definida (opcional)

p.e.:

```
struct   catalogo
{        char    nombre[20];
         char    titulo[10];
         int     numero; }  tarjeta;
```

Nombre de estructura (catalogo)

Nombre de variable (tarjeta)

Cada uno de los tipos de datos que puede contener una estructura se denomina campo de la estructura. Para acceder a cualquier campo de una estructura, se debe especificar el nombre de la variable y el nombre del campo en particular separados por el operador punto (.)

p.e. tarjeta.numero = 21;
tarjeta.nombre = "Antonio";

Una vez definida una estructura, se pueden definir más variables del tipo definido

```
struct catalogo libro, lista;
```

También es posible definir arrays de estructuras

```
struct catalogo guia[20]; //guia es un array de estructuras
```

para acceder a un campo de una estructura de un array, se haría:

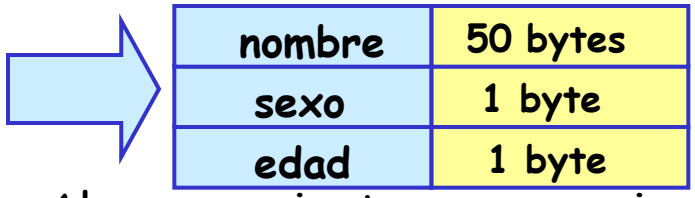
```
if (guia[12].titulo[0]=='Z') guia[12].numero=28;
```

Las estructuras se pueden pasar a las funciones como cualquier otro tipo de datos. Una función también puede devolver una estructura.

Cuando se pasa una estructura a una función, el paso se realiza mediante una llamada por valor, quedando la estructura de la llamada sin cambios.

También se pueden asignar valores de una estructura a otra de manera perfectamente válida. La inicialización sigue las mismas reglas:

```
p.e.    struct   ejemplo
        {
                 char     nombre[50];
                 char     sexo;
                 int      edad;
        } var[2] = {"Manuel",'V', 27, "Ana",'M',45);
```



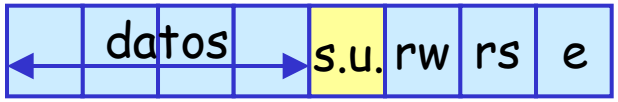
Almacenamiento en memoria

Se ha definido un array de 2 componentes tipo estructura ejemplo y se les ha asignado un valor inicial

Empleo de Estructuras para acceso a bits

Las estructuras pueden ser útiles para permitir el acceso a un bit individual dentro de un byte. Se definen campos de bits y se puede hacer referencia a cada uno de ellos de modo individual. En las estructuras, tras el identificador de campo y con dos puntos (:), se puede especificar el tamaño en bits que ocuparía ese campo (de 1 a 8)

```
p.e.:    struct   pines_LCD //esta estructura ocupa un  byte
         {
                  boolean  enable;
                  booelan  rs;
                  boolean  rw;
                  boolean  sin_uso;
                  int      datos : 4;
         } control;
```



Organización del byte

se puede hacer referencia luego a cada bit de modo individual

```
control.enable=1;          control.datos=7;
```

Punteros a Estructuras

El empleo de punteros a estructuras puede facilitar el acceso a éstas. Los punteros a estructuras se definen de la misma manera que a otros tipos de datos

p.e.:
```
struct   temp
{        int i;
         char ch;
} p, *q; //q es un puntero a la estructura
...
q=&p; //Ahora q apunta a la variable p (estructura)
```

si se quiere acceder a un elemento de la estructura mediante el puntero, se debe utilizar el operador flecha (->)

```
q->i = 10; //Asignación al elemento i de la estructura apuntada
```

se facilita el trasvase de información en las llamadas a funciones si en lugar de hacer la llamada con la estructura se hace con un puntero

Uniones

Una unión es una posición de memoria única que es compartida por una o más variables. Las variables que comparten una unión pueden ser de diferentes tipos de datos, pero sólo se puede emplear una variable en cada momento.

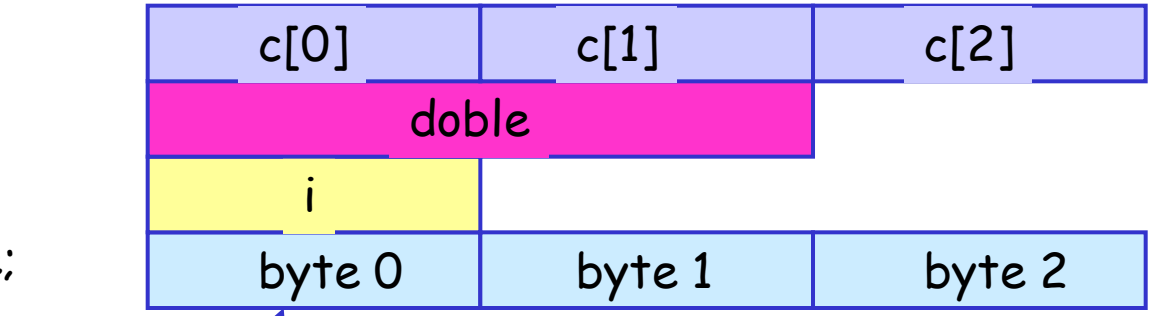
La definición de una unión se parece mucho a una definición de estructura

```
union especificador_tipo
{
        tipo campo_1;
        tipo campo_2;
        ...
        tipo campo_n;
} lista_variables;
```

Cuando se declara una unión, el compilador crea unas variables lo suficiente-mente grandes para guardar el tipo más grande de los elementos de la unión que han sido declarados

Ejemplo de declaración de unión y puntero:

```
union     comparte
{
          int       i;
          char      c[3];
          int16     doble;
} temp,*apunta;
```



Disposición en memoria de la variable temp y del puntero

El acceso a los distintos miembros de una unión se realiza con los mismos operadores que las estructuras: punto (.) y flecha (->)

```
          temp.doble      permite acceder a los dos bytes
          temp.c[2]       acceso a un byte del array
```

mediante punteros:

```
          apunte->i       acceso al byte
```

6.- Funciones

Las funciones son los bloques constructivos fundamentales en C. Todas las sentencias deben encontrarse dentro de funciones.

Las funciones deben ser declaradas o bien definidas antes de que sean llamadas o utilizadas (igual que en el caso de las variables)

Forma general de definición de una función:

```
especificación_tipo  nombre_función(tipo param1, tipo param2,..)
{
        cuerpo de la función (sentencias)
}
```

Declaración de prototipo de función (se anticipa que se va a definir luego)

```
especificación_tipo  nombre_función(lista de parámetros);
```

• Toda función en C puede devolver un valor a la sentencia o función que la llama. El valor devuelto se puede utilizar a la derecha del signo igual (=) en las sentencias de asignación

• Especificación_tipo es el que indica el tipo de dato que devuelve la función, si no se especifica nada se supone que la función devuelve un entero.
Las funciones pueden devolver cualquier tipo de dato salvo un array.
Si la función no devuelve ningún valor, la especificación debe ser tipo void

• La lista de parámetros es la lista de nombres de variables separados por comas con sus tipos asociados que recibirán los valores de los argumentos cuando se llame a la función. Una función puede no tener parámetros, en cuyo caso la lista estará vacía y se ponen sólo los paréntesis o bien la palabra reservada void

• En C todas las funciones están al mismo nivel de ámbito, esto quiere decir que no se puede definir una función dentro de otra función

• La manera que tiene una función para devolver un valor es mediante la sentencia return (para los compiladores de CCS):

```
return (expresión);
```

o bien

```
return expresión;
```

la expresión debe proporcionar el mismo tipo de dato que el especificado en el encabezamiento de la función. Si no debe devolver nada, se finaliza con:

```
return;
```

• Cuando una función se encuentra con una sentencia return, se vuelve a la rutina de llamada inmediatamente y las sentencias posteriores a return no serían ejecutadas

• Además de con la sentencia return, hay otra forma más en la que una función pude terminar su ejecución y retornar al lugar que la llamó: con la llave } de cierre de la función tras ejecutar la última sentencia de la misma

Ejemplo de prototipo, llamada y definición de una función:

```
//Prototipo que devuelve un entero y que recibe 3 enteros como argumentos
        int     volumen(int x, int y, int z);
                .........
        main( )
        {
                int vol;
                .....
                vol=volumen(5,7,12);        //llamada a la función
                ....
        }

//Definición de la función
        int     volumen(int x, int y, int z)
        {
                return(x*y*z);
        }
```

Argumentos de las funciones

Un argumento de una función es un valor que se pasa a ésta cuando la función es llamada. Una función puede no tener argumentos.

Cuando se define una función, se deben declarar unas variables especiales que reciban los argumentos de la llamada. Esas variables son las que aparecen a continuación del nombre de la función y entre paréntesis y se denominan parámetros formales

p.e.

```
int     suma(int a, int b)
{
        return(a+b);
}
main( )
{
        c = suma(10,23);
}
```

Parámetros formales

Argumentos de llamada

Se pueden pasar los argumentos a las funciones de dos formas:

Llamada por valor

Con este método se copia el valor del argumento de llamada en el parámetro formal de la función con lo que los cambios en los parámetros formales no afectarían a la variable usada en la llamada

p.e.

```
int     cuad(int x)
{
        return(x*x);
}
main( )
{
        int t=10, j;
        j=cuad(t);   //j toma el valor 100 pero t sigue siendo 10
}
```

la llamada por valor es la más habitual en los programas en C

Llamada por referencia

Es la segunda forma de pasar argumentos a una función. En este caso se copia la dirección del argumento en el parámetro formal de la función.

Dentro de la función, el parámetro formal se emplea para acceder al argumento usado en la llamada. Por tanto los cambios hechos a los parámetros afectan a la variables usada en la llamada. Las llamadas por referencia se realizan pasando punteros a las funciones, se pasan como cualquier otro valor habiéndolos declarado previamente.

p.e.
```
void      intercambia(int *x, int *y)
{ //función que intercambia el valor de dos argumentos enteros
          int temp;                    temp=*x;
          *x=*y;                       *y=temp;
}
main( )
{    int a=10, b=20;
     intercambia(&a, &b); //Se toman las direcciones de a y b y se pasan
}
```

Llamada por referencia (II): arrays como argumentos

Cuando se usa un array como argumento de una función, sólo se pasa la dirección del primer elemento (el puntero a ese elemento) y no una copia del array entero (recordemos que en C un array sin índice es un puntero al primer elemento

Hay dos formas de declarar un parámetro que va a recibir un puntero a array:

- Declarándolo como array: el compilador lo convierte a puntero — `void mostrar(int num[5])`
- Declarándolo como puntero: — `void mostrar(*int num)`

la llamada por referencia sería por paso del puntero

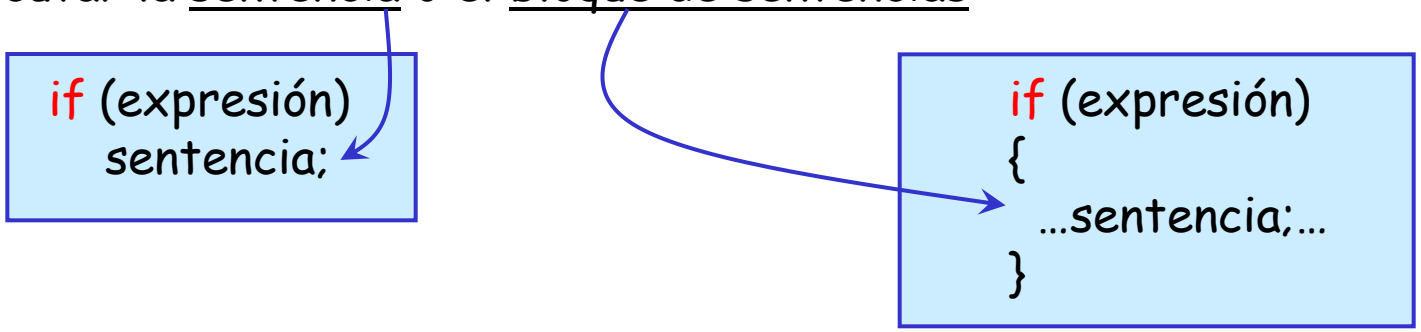
```
main( )
{       int t[5];…
        mostrar(t);
}
```

7.- Sentencias de control de programa

Estas sentencias son las que el lenguaje C utiliza para controlar el flujo de ejecución del programa. Los operadores lógicos y relacionales condicionan en muchos casos estas sentencias de control

Sentencia if

Se ejecuta una sentencia o bloque de código si la expresión que acompaña a if tiene un valor distinto de cero (verdadero), si es cero (falso) continúa sin ejecutar la sentencia o el bloque de sentencias



p.e.
```
if ((x==y)&&(t>=20)||(z==12))
        cuenta=0;
```

Sentencia if-else

Se evalúa una expresión y si es cierta, se ejecuta el primer bloque de código, (o sentencia 1) si es falsa se ejecuta el segundo bloque (o sentencia 2):

```
if (expresión)
{
        sentencias bloque 1;…
}
else
{
        sentencias bloque 2;…
}
```

```
if (expresión)
        sentencia 1;
else
        sentencia 2;
```

p.e.:

```
if (num>0)
        printf("numero positivo.\n");
else
        printf("numero negativo.\n");
```

Se pueden combinar varios if-else para establecer muchos caminos de decisión:

```
if (expresion1)
        sentencia(s);
else    if (expresion2)
                sentencia(s);
        else
                sentencia(s);
```

p.e.

```
if (numero==1)
        printf("es uno\n");
else    if (numero==2)
                printf("es dos\n");
        else    if (numero==3)
                        printf("es tres\n");
                else
                        printf("no es ninguno\n");
```

El operador ?

Este operador es realmente una abreviatura de la sentencia if-else. La sintáxis para sentencias de asignación es:

```
variable= (expresión1) ? (expresión2) : (expresión3);
```

se evalúa la expresión1 y si es cierta (no es cero), entonces se evalúa la expresión2 y se le asigna a variable, si es falsa (es cero) se evalúa la expresión3 y se le asigna a variable

p.e.

```
iabsoluto = i>0 ? i : (-i);
```

También se puede usar para otro tipo de sentencias:

```
(expresión1) ? (expresión2) : (expresión3);
```

p.e. `k ? j=0 : j=1;` `//Si k no es cero, entonces j=0 si no j=1`

Sentencia switch

Se emplea para sustituir a if-else cuando se trata de realizar una selección múltiple que compara una expresión con una lista de constantes enteras o caracteres. Cuando se da una coincidencia, el cuerpo de sentencias asociadas a esa constante se ejecuta hasta que aparezca break. La sintáxis sería:

```
switch (expresión)
{
case constante1:
        grupo1 de sentencias;
        break;
case constante2:
        grupo2 de sentencias;
        break;
...
default:
        grupo de sentencias;
}
```

break es opcional, si no aparece se sigue con el "case" siguiente

default es opcional y el bloque asociado se ejecuta sólo si no hay ninguna coincidencia con las constantes

No puede haber dos constantes iguales en dos "cases"

Ejemplo de sentencia switch:

```
char dia;          int orden;
...
switch (dia)
{
case 'L': dia=1;
          break;
case 'M': dia=2;
          break;
case 'X': dia=3;
          break;
case 'J': dia=4;
          break;
case 'V': dia=5;
          break;
case 'S': dia=6;
          break;
case 'D': dia=7;
          break;
default: dia=0;
}
```

Sentencia de bucle for

Se emplea para repetir una sentencia o bloque de sentencias.

```
for (inicialización; condición; incremento)
{
        sentencia(s);
}
```

En la inicialización se le asigna un valor inicial a una variable que se emplea para el control de la repetición del bucle (hay que declararla como variable), esa inicialización se ejecuta una sola vez.

La condición se evalúa antes de ejecutar la sentencia o bloque del bucle, en la expresión entrará normalmente la variable de control de repetición. Si la condición es cierta se ejecuta el bucle, si no se sale del mismo y se continúa con el resto del programa

El incremento se utiliza para establecer cómo cambia la variable de control cada vez que se repite el bucle

Ejemplo de bucle con for:

```
int       x;
for (x=1; x<=20; x++)
          printf("%d",x); //"Imprime" los números del 1 al 20
```

la sección incremento puede hacer referencia a cualquier otra asignación, no tiene por qué ser un incremento de la variable

p.e.
```
for (x=100; x>0; x--)
for (cuenta=0; cuenta<50; cuenta+=5)
```

También es posible anidar bucles para modificar dos ó mas variables de control

p.e.
```
int matriz[5][11];          int j,k; //Declaración de array e índices

for (j=0; j<5; j++)
          for(k=10; k>=0; k--)
                    matriz[j][k]=j*k; //Asignación a los elementos
```

Sentencia de bucle while

También se emplea para repetir una sentencia o bloque de sentencias. Ahora se realiza la repetición mientras sea cierta (no nula) una expresión

```
while (expresión)
{
        sentencia(s);
}
```

la expresión se evalúa antes de cualquier iteración. Si es falsa ya no se ejecuta la sentencia o bloque

p.e.
```
k=0;        char tira[20];
//Fragmento que cuenta en k el número de caracteres de una tira
while (tira[k]) //la tira acabaría con null (0)
{
        k++;
}
```

Sentencia de bucle do-while

Tercer tipo de sentencia de repetición:

```
do
{
        sentencia(s);
}
while (expresión)
```

ahora las sentencias se ejecutan antes de que se evalúe la expresión. Por tanto el bucle se ejecuta siempre al menos una vez

p.e.

```
char letra;
do
{
        letra = getchar( ); //función que recoge un carácter
}
while (letra != 'A'); //Este bucle espera hasta que reciba una 'A'
```

Sentencia break

Esta sentencia tiene dos usos posibles:

a).- Permite salir de un bucle de repetición en cualquier punto del bloque. Si se encuentra esta sentencia, el programa pasa a la siguiente sentencia tras el bucle

p.e.:

```
k=12;
for (i=0; i<=25; i++)
{
        printf("%d",i);
        if (i==k)
                break; //"imprime" desde 0 hasta k=12
}
```

b).- Se puede utilizar para finalizar un "case" en una sentencia switch

p.e.:

```
...
case 'M':
        dia=2;
        break;
```

Sentencia continue

Funciona de manera parecida a la sentencia break dentro de un bucle. Sin embargo, en lugar de forzar la terminación del mismo continue fuerza una nueva iteración del bucle y salta el código que hay hasta que se evalúa la condición del bucle

p.e.

```
char *cadena; //definimos puntero a cadena
int espacios;
...
for(espacios=0; *cadena; cadena++)
{
        if (*cadena !=' ') continue;
        //si carácter no es blanco sigue a la condición e incremento
        espacios++;
        //solo llega a incrementar espacios si era blanco el carácter
}
//Este fragmento cuenta por tanto los espacios en una cadena
//de caracteres
```

Sentencia return

Esta sentencia se utiliza para finalizar y volver desde una función hacia el punto en que se le llamó. Se puede devolver un valor a la función que hizo la llamada:

`return (expresión);` o bien `return expresión`

si se omite la expresión, el valor devuelto estaría indefinido (no hay error). Si no se necesita devolver ningún valor, lo más correcto sería declarar el tipo void en la declaración del valor devuelto por la función

Si no se incluye una sentencia return en una función, ésta se finaliza tras la ejecución de la última línea del código

p.e.
```
void    no_hace_nada(int c)
{       c++;
        return;
}
```

Sentencia goto

El uso del goto tiene muy mala imagen entre los programadores de alto nivel ya que contribuye a hacer ilegibles los programas. En C es posible escribir los códigos sin su uso, sin embargo vamos a indicar su existencia y si alguien lo necesita, que lo use si no puede evitarlo (lo usamos en ensamblador).

La sentencia goto necesita una etiqueta para operar, una etiqueta es cualquier identificador válido en C seguido de dos puntos (:). La etiqueta debe estar en la misma función que el goto (¡ no se puede saltar entre funciones !)

Sintáxis

```
goto etiqueta;
...
etiqueta: sentencia;
```

p.e.:

```
bucle:
        x++;
        if (x<=100) goto bucle;
```

8.- C específico para los PIC

Tras haber recorrido los elementos básicos del C (más o menos estándar), se deben conocer las configuraciones particulares, funciones, operaciones y directivas del compilador específico con el que se va a trabajar.

Las principales diferencias entre compiladores residen en las directivas (*pre-processor commands*) y en las funciones integradas (*built-in functions*)

En ocasiones será necesario insertar instrucciones de ensamblador en medio de un programa en C para compactar código, reducir tiempos de ejecución o porque se desea que la rutina sea tal cual. En esos casos, se utilizan las directivas #ASM y #ENDASM para delimitar el inicio y el final de código en ensamblador

Los compiladores de CCS presentan un completo abanico de funciones integradas para reducción del tiempo de desarrollo en las aplicaciones. Las funciones están orientadas al manejo del núcleo y de los diversos módulos internos de los microcontroladores PIC

Directivas de Preprocesado para compiladores de CCS

Pre-Processor Command Summary			
Standard C		**Device Specification**	
#DEFINE IS STRING	24	#DEVICE CHIP	25
#ELSE	28	#ID NUMBER	27
#ENDIF	28	#ID "filename"	27
#ERROR	26	#ID CHECKSUM	27
#IF expr	28	#FUSES options	27
#IFDEF id	29	#TYPE type=type	41
#INCLUDE "FILENAME"	30	**Built-in Libraries**	
#INCLUDE <FILENAME>	30	#USE DELAY CLOCK	42
#LIST	34	#USE FAST_IO	42
#NOLIST	34	#USE FIXED_IO	43
#PRAGMA cmd	38	#USE I2C	43
#UNDEF id	41	#USE RS232	44
Function Qualifier		#USE STANDARD_IO	45
#INLINE	31	**Memory Control**	
#INT_DEFAULT	32	#ASM	19
#INT_GLOBAL	33	#BIT id=const.const	22
#INT_xxx	31	#BIT id=id.const	22
#SEPARATE	40	#BYTE id=const	22
Compiler Control		#BYTE id=id	22
#CASE	23	#LOCATE id=const	34
#OPT n	35	#ENDASM	19
#PRIORITY	38	#RESERVE	39
#ORG	35	#ROM	39
		#ZERO_RAM	46
		Pre-Defined Identifier	
		__DATE__	24
		__DEVICE__	26
		__PCB__	37
		__PCM__	37
		__PCH__	38

Directivas de preprocesado más habituales:

#ASM #ENDASM	Las líneas entre estas dos directivas deben ser instrucciones en ensamblador que se insertan tal y como aparecen
#BIT id=x.y	Se crea una variable tipo bit correspondiente al bit y del byte x en memoria. p.e.: BIT TOIF=0xb.2
#BYTE id=x	Se crea una variable y se sitúa en el byte x en memoria p.e.: #BYTE PORTB=0x06. Si ya existía esa variable se coloca físicamente en la posición especificada
#DEFINE id texto	El identificador se sustituye por el texto adjunto p.e.: #DEFINE BITS 8
#DEVICE chip	Define el micro para el que se escribe el código p.e.: #DEVICE PIC16F877
#FUSES options	Define la palabra de configuración para la grabación

#INCLUDE <fichero> #INCLUDE "fichero"	Se incluye el texto del fichero especificado en el directorio o fuera de él
#INLINE	La función que sigue a este directiva debe ser copiada en memoria de programa cada vez que se le llame. Puede servir para salvar posiciones de stack y mejorar velocidad
#INT_xxxx	Indica que la función que viene a continuación es un programa de tratamiento de una interrupción tipo xxxx
#INT_GLOBAL	Indica la función que sigue es programa de tratamiento de interrupción genérico, no se incluye código de salvaguarda de registros ni de recuperación como con INT_xxxx
#LIST #NOLIST	Conmutan generación o no generación de líneas en el fichero de listado (.LST)
#ORG start	Sirve para situar el código a partir de una determinada posición de la memoria de programa

#PRIORITY ints Se emplea para establecer un orden de prioridad en las interrupciones, ints es una lista de interrupciones posibles
p.e.: #PRIORITY rtcc,rb

#ROM dir={lista} Sirve para insertar datos en el fichero .HEX. También en EEPROM de datos.
p.e.: #ROM 0x2100={1,2,3,4,5}

#SEPARATE Indica que el procedimiento que sigue a la directiva debe implementarse de manera separada, no INLINE copiando el código varias veces. De esta manera se ahorra ROM

#USE DELAY (clock=frecuencia en Hz)

Define la frecuencia del oscilador que se va a utilizar y se emplea para realizar los cálculos para las funciones integradas de retardo (delay_ms( ) y delay_us( ))
p.e.: #USE DELAY (clock=4000000)

#USE FAST_IO(puerto)
#USE FIXED_IO(puerto_oputputs=pin, pines)
#USE STANDARD_IO(puerto)

Directivas empleadas para indicar el compilador, cómo debe generar el código para las instrucciones de E/S

#USE I2C(opciones)

Directiva que indica modo de trabajo para las funciones de implementación de comunicación en bus I2C

#USE RS232(opciones)

Especifica la configuración de una comunicación serie de E/S: relación de baudios, pines de transmisión y recepción, etc.

Funciones Integradas para Compiladores de CCS (I)

BUILT-IN FUNCTIONS

Built-In Function List By Category			
RS232 I/O		**Parallel Slave I/O**	
getc()	69	setup_psp()	108
putc()	90	psp_input_full()	89
gets()	70	psp_output_full()	89
puts()	91	psp_overflow()	89
printf()	88	**Delays**	
kbhit()	78	delay_us()	64
set_uart_speed()	102	delay_ms()	64
I2C I/O		delay_cycles()	63
i2c_start()	72	**Processor Controls**	
i2c_stop()	73	sleep()	117
i2C_read	71	reset_cpu()	95
i2c_write()	74	restart_cause()	95
i2c_poll()	71	disable_interrupts()	65
Discrete I/O		enable_interrupts()	66
output_low()	85	ext_int_edge()	67
output_high()	85	read_bank()	92
output_float()	84	write_bank()	125
output_bit()	83	**Bit Manipulation**	
input()	74	shift_right()	115
output_X()	86	shift_left()	114
input_X()	75	rotate_right()	98
port_b_pullups()	87	rotate_left()	97
set_tris_X()	101	bit_clear()	60
SPI two wire I/O		bit_set()	61
setup_spi()	109	bit_test()	62
spi_read()	118	swap()	123
spi_write()	119	**Capture/Compare/PWM**	
spi_data_is_in()	117	setup_ccpX()	104
		set_pwmX_duty()	99

Funciones Integradas para Compiladores de CCS (II)

Built-In Function List By Category... Continued			
Timers		**Standard C Char**	
setup_timer_X()	109	atoi()	59
set_timer_X()	100	atol()	59
get_timer_X()	69	tolower()	124
setup_counters()	106	toupper()	124
setup_wdt()	113	isalnum()	77
restart_wdt()	96	isalpha()	77
A/D Conversion		isamoung()	76
setup_adc_ports()	103	isdigit()	77
setup_adc()	103	islower()	77
set_adc_channel()	103	isspace()	77
read_adc()	91	isupper()	77
Analog Compare		isxdigit()	77
setup_comparator()	105	strlen()	120
Internal EEPROM		strcpy()	123
read_eeprom()	94	strncpy()	120
write_eeprom()	126	strcmp()	120
read_program_eeprom()	94	stricmp()	120
write_program_eeprom()	126	strncmp()	120
read_calibration()	93	strcat()	120
Standard C Math		strstr()	120
abs()	59	strchr()	120
acos()	59	strrchr()	120
asin()	59	strtok()	120
atan()	59	strspn()	120
ceil()	62	strcspn()	120
cos()	63	strpbrk()	120
exp()	67	strlwr()	120
floor()	68	**Standard C memory**	
labs()	79	memset()	83
log()	81	memcpy()	82
log10()	81	**Voltage Ref**	
pow()	87	setup_vref()	113
sin()	116		
sqrt()	119		
tan()	116		