

# **Cosas a saber antes de comenzar a programar microcontroladores en C.**

Autor: Agustín Gimeno

Version: 1.0

Información de contacto: [agustin.gimeno@gmail.com](mailto:agustin.gimeno@gmail.com)

## **Prólogo.**

El presente apunte tiene por objetivo ser una guía de estudio de acceso sencillo y rápido para reforzar aquellos conceptos ya vistos en la asignatura Informática I de los cuales se hará uso en Informática II, además de dar una introducción a distintos temas que luego serán profundizados en Digitales I principalmente, enfocado a la programación de microcontroladores.

Por lo tanto va dirigida y se recomienda su lectura a aquellos alumnos sin formación técnico electrónica o similar (electrotécnico, mecatrónico, etc).

## **Índice de temas:**

### 1-Sistemas de numeración.

1.1-Teorema fundamental de numeración. Sistema Decimal, Binario, Hexadecimal.

1.2-Conversión entre Binario, Decimal y Hexadecimal.

### 2-Operadores lógicos.

2.1-Compuertas lógicas, tablas de verdad y operadores en C.

### 3-GPIO Microcontroladores.

3.1-Introducción.

3.2-Máscaras.

### 4-Herramientas en C.

4.1-Tipos de datos soportados en C51(Keil).

4.2-Tipos de memoria.

4.3-Definición de variables.

4.4-Macros.

4.5-Campos de bits y uniones.

4.6-Typedef.

4.7-Generando Tablas constantes en memoria de programa.

## 1-Sistemas de numeración.

Un sistema de numeración es un conjunto de símbolos que siguen reglas para poder así generar todos los números válidos. En particular, los más interesantes para la programación de microcontroladores son el Binario y el Hexadecimal. Sin embargo se comenzará explicando el Decimal por ser con el cual todas las personas se encuentran familiarizadas. El correcto manejo del sistema de numeración Binario y la conversión a Hexadecimal le dará mayor destreza al programar microcontroladores.

### 1-1 Teorema fundamental de numeración. Sistema Decimal, Binario, Hexadecimal.

Un sistema de numeración posicional se encuentra descrito por los siguientes componentes: base, símbolo numérico y posición del dígito o potencia.

La base es lo que le da nombre al sistema de numeración. Entonces:

- para el sistema decimal la base es 10.
- para el sistema binario la base será 2.
- para el sistema hexadecimal la base será 16.

A su vez, la base también indica la cantidad de símbolos distintos con que contará el sistema:

- sistema decimal: 10 símbolos distintos, [0,1,2,3,4,5,6,7,8,9].
- sistema binario: 2 símbolos distintos, [0,1].
- sistema hexadecimal: 16 símbolos distintos, [0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F].

Por lo tanto, una forma para calcular el valor de un número, es decir pasarlo a decimal, sería descomponerlo de la siguiente forma.

$$N = d_n b^n + \dots + d_1 b^1 + d_0 b^0 .$$

Estudiemos esto primero desde el punto de vista de cómo se forma un número decimal y luego veremos cómo convertir un número de otra base a este sistema.

Supongamos el número: 1234(10). Todavía no habíamos hablado nada acerca de la posición del dígito o potencia, pero aquí queda más que en evidencia a qué nos estamos refiriendo. Para el ejemplo, el número 4 se encuentra en la posición 0, el número 3 en la posición 1, el número 2 en la posición 2 y así sucesivamente. Por lo tanto. el número 1234 en base decimal se puede escribir también como:

$$N = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 = 1234$$

Se puede observar entonces que la base "b" se reemplazó por 10, el sistema de numeración en el cual estaba el número, y luego se fueron

colocando los distintos símbolos a la potencia correspondiente según la posición.

Si quisiéramos averiguar el valor de un determinado número binario no tendríamos más que seguir el mismo procedimiento, colocando "0" o "1" en "d", la base "b" en 2 y la potencia según corresponda. Sin embargo, realizar el cálculo de esta manera lleva mucho tiempo. Al final de este capítulo se darán algunas técnicas ágiles de conversión.

El sistema binario utiliza entonces como sus dos únicos símbolos cero y uno (0 y 1), haciéndolo el sistema ideal para ser representado por circuitos electrónicos.

El sistema hexadecimal se utiliza en informática y en la programación de microcontroladores como una forma más práctica y prolija de escribir los números. Cuando veamos las formas ágiles de conversión explicaremos por qué.

A continuación se muestra una tabla con la equivalencia entre Decimal, Binario y Hexadecimal.

Decimal	Binario	Hexa
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

## 1.2-Conversión entre Binario, Decimal y Hexadecimal.

### Conversión ágil de decimal a binario y viceversa.

Existen diversos métodos para realizar la cuenta, pero el que generalmente es más rápido consiste en ir sumando potencias de 2. Por lo tanto, primero vamos a escribir todas las potencias de 2 que un estudiante de ingeniería electrónica no puede dejar de saber:

65536,32768,16384,8192,4096,2048,1024,512,128,64,32,16,8,4,2,1.

Esto significa que con un dígito podemos escribir dos combinaciones, con dos podremos escribir cuatro combinaciones y así sucesivamente; entonces si puedo escribir 4 combinaciones tengo un rango de [0,1,2,3].

Ahora ya estamos en condiciones de ver cuál es la lógica para hacer una rápida conversión. Se toma el número y se busca la potencia de 2 más próxima inferior, se escribe un uno (1). Luego se suma la próxima potencia de 2 (obviamente inferior) si la suma se pasa del número se anota un cero (0), sino un uno (1); se continúa de la misma forma en orden descendente hasta que se obtiene el número. Una vez que se obtiene, se completa con ceros el resto de las posiciones. Recomendación: las primeras veces se recomienda escribir todas las potencias de 2 anteriores e ir colocando los ceros y los unos debajo.

Veamos un ejemplo. Convertir el número 1025. Comenzamos buscando la potencia de dos inferior más cercana, en este caso 1024. Entonces, por ser la primera vez, escribimos todas las potencias inferiores y 1024.

1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.

1

Luego comenzamos a sumar, por supuesto que 1024+ 512 superará a 1025, de hecho todas las potencias excepto 1 lo harán. Por lo tanto, llenamos todas las potencias excepto 1 con ceros debajo, y finalmente con un 1 debajo de la potencia de 2 1.

1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.

1 0 0 0 0 0 0 0 0 0 1.

Veamos ahora que pasa en el caso que tuviéramos el número 1312. Se puede ver que la potencia inmediata inferior es 1024. Así que nuevamente procedemos de la misma manera, primero escribimos todas las potencias, de esta forma ya las vamos memorizando. Luego este cálculo lo haremos mentalmente. Después sumamos 1024+512 y como nos pasamos de 1312 colocamos un cero. Hasta ahora tenemos:

1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.

1 0

Ahora pasamos a sumar  $1024+256=1280$ , es menor al número 1312. por lo tanto debajo de 256 colocaremos un uno (1).

1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.  
 1 0 1

Proseguimos de la misma manera hasta encontrar que al sumar 32 obtenemos el número 1312, por lo tanto colocamos un 1 debajo de 32 y el resto lo completamos con ceros.

1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.  
 1 0 1 0 0 1 0 0 0 0 0.

Por supuesto si sumamos todas las potencias de dos desde un determinado número, obtendremos la potencia siguiente menos 1, para nuestro ejemplo si sumamos 1024 y todas las potencias inferiores obtendremos  $2048-1$ , o sea 2047.

De la misma forma si queremos convertir en forma rápida un número binario a decimal, arrancamos desde la derecha y comenzamos a sumar potencias de 2 donde haya un 1. Por ejemplo:

1 1 1 1      1 0 1 0  
 $8 + 4 + 2 + 1 = 15$      $8 + 0 + 2 + 0 = 10$

En realidad, es aplicar la fórmula de más arriba, sólo que explicado en forma más sencilla.

### Conversión ágil entre binario y hexadecimal.

Para convertir rápidamente de binario a hexadecimal, se toman grupos de 4 dígitos binarios y se los convierte primero a decimal y luego a hexadecimal. El sistema decimal y hexadecimal hasta el 9 inclusive son exactamente iguales, luego se rellena con A,B,C,D,E,F para las posiciones de 10 a 15. Por eso es importante saber la tabla anterior. Por ejemplo:

11111111 = 1111-1111(binario)  
                   15      15    (decimal)  
                   F      F    (hexadecimal)  
 10101010=1010-1010(binario)=10 -10(decimal)=AA(hexadecimal)





De esta forma se evita en un programa escribir tantos ceros y unos que dificultan la lectura e incluso inducen a errores. En un sistema de 8 bits como el del 8051 no suele ser tan crítico, pero si lo pensamos desde el punto de vista de un microcontrolador de 32 bits las cosas cambian.

Probablemente se preguntarán por qué se toman 4 posiciones, y no 3 o 5. Con 4 dígitos en forma binaria, se pueden representar hasta 16 combinaciones distintas que es justamente la base del sistema hexadecimal, por lo tanto 4 bits pueden albergar sólo un dígito hexadecimal, haciéndolo idealmente práctico para nuestro uso.

A modo de resumen, los puntos sobresalientes que deberán ser manejados con fluidez son la tabla de conversión entre decimal, binario y hexadecimal del 0 al 15, las potencias de 2 hasta la posición 16 preferentemente y los métodos de conversión entre números. Si bien es cierto que hoy en día casi cualquier calculadora, celular, etc, tiene alguna herramienta para convertir entre sistemas de unidades es primordial entender el sistema binario para todo lo que son operaciones lógicas y el tema que se desprende de esto último: MÁSCARAS.

## 2-Operadores Lógicos

### 2-1-Compuertas lógicas, Tablas de Verdad y Operadores en C a nivel bit.

Tipo	Símbolo	Operador en C	Tabla de verdad		
NOT		$\sim$ (ALT+126)	Entrada		Salida
			A		NOT A
			0		1
			1		0
AND		& (ALT+38)	Entrada		Salida
			A	B	A AND B
			0	0	0
			0	1	0
			1	0	0
			1	1	1
OR		 (ALT+124)	Entrada		Salida
			A	B	A OR B
			0	0	0
			0	1	1
			1	0	1
			1	1	1
XOR		^ (ALT+94)	Entrada		Salida
			A	B	A XOR B
			0	0	0
			0	1	1
			1	0	1
			1	1	0

Otros operadores a nivel bit que posee el lenguaje C son los operadores de desplazamiento, o "Shift Operators", "<<" y ">>".



De esta forma los 6 operadores a nivel bit que posee el lenguaje C son:

Symbol	Operator
&	bitwise AND
	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

Estos operadores sólo se pueden utilizar para realizar operaciones entre números enteros, obviamente del mismo tipo, es decir longitud de bits.

Para aquellos que nunca realizaron una operación lógica se recomienda leer los siguientes ejemplos y realizar los ejercicios que se plantean para ganar práctica, ya que es algo que no se dejará de utilizar a lo largo de la materia.

Ejemplo 1:

	AND (&)
A	1 1 0 0 1 0 1 0
B	0 0 1 1 1 0 1 1
Z	0 0 0 0 1 0 1 0

Ejemplo 2:

	OR ( )
A	1 1 0 0 1 0 1 0
B	0 0 1 1 1 0 1 1
Z	1 1 1 1 1 0 1 1

Ejemplo 3:

	XOR (^)
A	1 1 0 0 1 0 1 0
B	0 0 1 1 1 0 1 1
Z	1 1 1 1 0 0 0 1

Ejercicios planteados:

Ejercicios 1:

	AND (&)
A	1 0 1 0 1 1 1 0
B	0 1 1 0 1 0 0 1
Z	

Ejercicio 2:

	OR ( )
A	1 0 1 0 1 1 1 0
B	0 1 1 0 1 0 0 1
Z	

Ejercicio 3:

	XOR (^)
A	1 0 1 0 1 1 1 0
B	0 1 1 0 1 0 0 1
Z	

**Es importante** no confundir los operadores a nivel bit con los operadores lógicos, AND(&&) y OR(||); estos últimos sirven para evaluar expresiones booleanas (donde sólo intervienen datos del tipo bool "true or false") como puede ser la condición para que se ingrese dentro de un "if " o se continúe realizando un determinado "while".

### 3-GPIO Microcontroladores.

#### 3-1 Introducción.

El medio para comunicar el microcontrolador con todo el hardware exterior es a través de sus entradas y salidas. Se entiende por GPIO a General Purpose Input/Output, entradas/salidas de propósito general. Las entradas y salidas entonces pueden ser de propósito general o especiales, comunicación serie, analógicas, etc. En la mayoría de los microcontroladores todas las salidas/entradas pueden ser de propósito general para luego configurarlas y transformarlas en especiales; pero no cualquier entrada/salida podrá transformarse en una especial.

La forma en que se agrupan las GPIO es a través de los puertos, como pueden ser PORT0, PORT1, PORTA, PORTB, etc y dependerá del fabricante. A su vez, cada puerto tiene asignada una determinada cantidad de pines que constituyen el nexo entre el hardware y el software o firmware que nosotros programamos; cada pin está conectado o es una pata del microcontrolador.

A lo largo de la cursada de Informática II se verá que las salidas del tipo especial estarán asociadas a un determinado periférico del microcontrolador, UART, I2C, PWM, ADC, DAC, Timmers/Counters, etc; y sólo se podrán utilizar determinados pines para la conexión.

Teniendo el concepto de lo que es un puerto y un pin se abordará ahora el concepto de máscara.

### **3-2 MASCARAS.**

Los pines del microcontrolador no siempre están disponibles para poder manipularlos (programarlos) individualmente, sino que muchas veces se debe trabajar sobre todo el puerto; dependerá del fabricante del microcontrolador si tenemos acceso a ellos individualmente o no. A esto se lo conoce como "bit-addressable" o "no bit-addressable".

Las máscaras son una herramienta muy útil para trabajar en forma rápida con datos que no son bit-addressable o cuando tenemos que trabajar sobre varios del mismo conjunto (Puerto) sin tocar el resto. Consiste en aplicar funciones lógicas (AND; OR, XOR) a nivel bit para poder leer/escribir datos sin modificar otros. Se termina entonces enmascarando los datos que nos interesa modificar de aquellos que no.

Un ejemplo de uso de máscaras podría ser el siguiente: un puerto cuyos pines no son bit-addressable, es decir que no lo podemos acceder individualmente, tiene todos sus pines configurados como salida, pero estas salidas no son todas utilizadas para lo mismo por lo que se desea acceder sólo a 3 pines de ella y luego a los otros 5 pines. (Se supone un puerto de 8 bits, es decir 8 pines). Se desean modificar entonces 3 pines pero manteniendo el valor de los otros 5.

Es importante entonces volver sobre el capítulo anterior y observar las tablas de verdad para concluir lo siguiente:

- Cualquier bit al cual se le haga una operación AND con cero (0) será "borrado" es decir pasará a valer cero, pero cualquier bit al cual le haga una operación AND con un uno (1) permanecerá en el estado en que estaba.

- Cualquier bit al cual se le haga una operación OR con un cero (0) no modificará el estado del mismo, pero si se hace una OR con un 1 el bit se "seteará" o pondrá en uno (1).
- Si se realiza un XOR de un bit cualquiera con un uno (1) se complementará el estado del bit anterior, mientras que con cero (0) permanece en el estado anterior.

De esta forma la metodología a emplear sería leer el dato/puerto, aplicarle 1 o 2 máscaras (con esto por lo general se puede resolver la mayoría de los trabajos) y volver a escribir el dato en el puerto o leer el dato directamente.

Ejemplo1: Se tienen 3 pines de entrada los cuales constituyen un dato y se quiere separar del resto. El puerto es de 8 pines, que queda representado en memoria por un byte. Por lo tanto lo que se busca es que en otra variable se tenga el dato de bits que se tiene a la entrada con los otros 5 bits en cero. La operación que se realizará es una AND con los 5 bits en cero (0) y los 3 bits que nos interesan en uno (1).

Dato obtenido del Puerto:    XXXXX101  
 MÁSCARA:                    0 0 0 0 0 111  
 Dato con la máscara:        0 0 0 0 0 101  
 Nota: "X" don't care o no importa.

Vemos entonces que pudimos recuperar la parte de la información que nos interesaba.

Ejemplo2: Se tiene ahora todo el puerto seteado como salida, 6 pines están conectados a leds de indicación mientras que los otros 2 se utilizan para comandar dos relés. Se desea cambiar el estado de los 6 leds al mismo tiempo sin afectar a los dos relés.

Procedimiento: se deberá leer el puerto con los estados de las salidas actuales y guardarlo en una variable, borrar el estado de los leds (aplicar una AND) para luego cargar el nuevo (aplicar una OR).

Dato del puerto:            101101XX  
 MÁSCARA1(AND):            000000 1 1  
 Dato con la máscara 1: 000000XX  
 Dato a cargar (OR):        110010 0 0  
 Dato resultante:            110010XX

Posteriormente, el dato resultante lo moveremos al puerto. Aclaración: entre "Dato con la máscara 1" y "Dato a cargar" se aplicó una "OR".

Importante: no se puede aplicar una OR directamente debido a que si en la nueva configuración de leds necesito tener un cero (0) donde antes tenía un uno (1) no lo podré lograr. Tampoco podré obtener un uno (1) si antes tenía un

(0) y aplico una AND directamente.

Con estos ejemplos se explican los casos de estudio más comunes en la aplicación de máscaras. Sin embargo, su uso es todavía más extenso y con la práctica se seguirá trabajando sobre el tema.

Otro uso de máscaras puede ser setear uno o mas pines de un puerto, usando máscaras individuales y aquí es importante saber manejar la tabla que se introdujo en el capítulo uno con las equivalencias entre binario, decimal y hexadecimal. Se puede observar que:

```
#define PRIMERBIT 01 /*00000001*/
#define SEGUNDOBIT 02 /*00000010*/
#define TERCERBIT 04 /*00000100*/
#define CUARTOBIT 08 /*00001000*/
```

Por lo tanto si queremos encender el segundo y tercer bit sin tocar el resto del puerto podemos hacer una máscara como sigue:

```
PORTA |= SEGUNDOBIT | TERCERBIT;
```

y si quisiéramos borrarlos podríamos:

```
PORTA&=~(SEGUNDOBIT | TERCERBIT);
```

Nota: a veces también se le llama máscara a un número que pertenece a un estado o a algún tipo de información útil dentro del software que por lo general queda declarada por un define.

## 4-Herramientas en C.

### 4.1-Tipos de datos soportados en C51(Keil).

Data Types	Bits	Bytes	Value Range
<a href="#">bit</a>	1		0 to 1
signed <a href="#">char</a>	8	1	-128 — +127
unsigned <a href="#">char</a>	8	1	0 — 255
<a href="#">enum</a>	8 / 16	1 or 2	-128 — +127 or -32768 — +32767
signed short <a href="#">int</a>	16	2	-32768 — +32767
unsigned short <a href="#">int</a>	16	2	0 — 65535
signed <a href="#">int</a>	16	2	-32768 — +32767
unsigned <a href="#">int</a>	16	2	0 — 65535
signed long <a href="#">int</a>	32	4	-2147483648 — +2147483647
unsigned long <a href="#">int</a>	32	4	0 — 4294967295
<a href="#">float</a>	32	4	±1.175494E-38 — ±3.402823E+38
<a href="#">double</a>	32	4	±1.175494E-38 — ±3.402823E+38
<a href="#">sbit</a>	1		0 or 1
<a href="#">sfr</a>	8	1	0 — 255
<a href="#">sfr16</a>	16	2	0 — 65535

## 4.2-Tipos de memoria.

Memory Type	Description
<a href="#">code</a>	Program memory (64 KBytes); accessed by opcode MOVC @A+DPTR.
<a href="#">data</a>	Directly addressable internal data memory; fastest access to variables (128 bytes).
<a href="#">idata</a>	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
<a href="#">bdata</a>	Bit-addressable internal data memory; supports mixed bit and byte access (16 bytes).
<a href="#">xdata</a>	External data memory (64 KBytes); accessed by opcode MOVX @DPTR.
<a href="#">far</a>	Extended RAM and ROM memory spaces (up to 16MB); accessed by user defined routines or specific chip extensions (Philips 80C51MX, Dallas 390).
<a href="#">pdata</a>	Paged (256 bytes) external data memory; accessed by opcode MOVX @Rn.

## 4.3-Definición de variables.

La definición de variables se hace a través del prototipo que se presenta más abajo, el cual tiene tres campos. El primero, tipoDeMemoria, es donde va a estar alojada la variable; si el programador no define este campo dejándolo en blanco la variable será posicionada en la memoria por defecto que tenga seleccionada el compilador. El segundo campo es tipoDeVariable; debido a que no se posee una gran cantidad de memoria ram hay que ser cuidadosos con el tipo de variable que se escoge pero, a su vez, se debe conocer exactamente cuál es el rango de valores que podría llegar a tomar la variable. Por último, el nombre de la variable debe ser elegido respetando las buenas prácticas de programación de código estándar, esto incluye también darle un nombre representativo a la variable en cuestión.

Prototipo de definición de una variable:

tipoDeMemoria tipoDeVariable NombreDeVariable;

Por ejemplo:

```
xdata      unsigned char    varEnxdata;
code unsigned int          varEnCode = 0xFAFA;
code unsigned char        arrayEnCode[3] = [1,3,5,6];
```

Nota: las variables en memoria de programa que usan la palabra reservada code deben ser inicializadas inline, se usan como constantes.

## 4.4-Macros.

El pre-procesador de C proporciona ciertas facilidades de lenguaje, conceptualmente es un primer paso separado en la compilación. Los dos elementos que se usan con más frecuencia son #include, para incluir el

contenido de un archivo durante la compilación, y `#define`, para reemplazar un símbolo por una secuencia arbitraria de caracteres. Otras características de interés son la compilación condicional y macros con argumentos.

En la siguiente tabla se muestran todas las directivas del pre-procesador:

Directive	Description
<a href="#"><code>#define</code></a>	Defines a preprocessor macro or constant.
<a href="#"><code>#elif</code></a>	Initiates an alternative branch of the if condition, when the previous <b><code>#if</code></b> , <b><code>#ifdef</code></b> , <b><code>#ifndef</code></b> , or <b><code>#elif</code></b> branch was not taken.
<a href="#"><code>#else</code></a>	Initiates an alternative branch when the previous <b><code>#if</code></b> , <b><code>#ifdef</code></b> , or <b><code>#ifndef</code></b> branch was not taken.
<a href="#"><code>#endif</code></a>	Ends a <b><code>#if</code></b> , <b><code>#ifdef</code></b> , <b><code>#ifndef</code></b> , <b><code>#elif</code></b> , or <b><code>#else</code></b> block.
<a href="#"><code>#error</code></a>	Outputs an error message defined by the user.
<a href="#"><code>#if</code></a>	Evaluates an expression for conditional compilation.
<a href="#"><code>#ifdef</code></a>	Evaluates an expression for conditional compilation. The argument to be evaluated is the name of a definition.
<a href="#"><code>#ifndef</code></a>	Same as <b><code>#ifdef</code></b> but the evaluation succeeds if the definition is not defined.
<a href="#"><code>#include</code></a>	Reads source text from an external file. The notation sequence determines the search sequence of the included files. The compiler searches for include files specified with less-than/greater-than symbols ('<', '>') in the include file directory. Include files specified with double-quotes (" ") are searched for in the current directory.
<a href="#"><code>#line</code></a>	Specifies a line number and an optional filename. This specification is used in error messages to identify the error position. For line synchronization with debug information or the listing file, this preprocessor directive must be combined with the <a href="#"><code>NOPROCESS</code></a> Compiler directive.
<a href="#"><code>#message</code></a>	Outputs a information message defined by the user.
<a href="#"><code>#pragma</code></a>	Allows you to specify directives that may be included on the compiler command line. Pragmas may contain the same directives that are specified on the command line.
<a href="#"><code>#undef</code></a>	Deletes a preprocessor macro or constant definition.
<a href="#"><code>#warning</code></a>	Outputs a warning message defined by the user.

#### 4.4.A-Substitución de macros sin argumentos.

La substitución de macros del tipo más sencillo, sin argumentos, conceptualmente consiste en un simple reemplazo de texto.

Una definición tiene la forma:

```
#define NOMBRE texto de reemplazo
#define ON 1
#define OFF 0
```

Cada ocurrencia dentro del código fuente de "ON" u "OFF" será

reemplazada por "1" y "0" respectivamente. Un error común que debe evitarse es colocar un ";" detrás del "1" y el "0" ya que estas definiciones no son líneas de procesamiento; lo que es peor el pre-procesador reemplazará "1;" de haberlo colocado.

Supongamos que se tiene la siguiente definición:

```
#define ON      1; (MAL!)
```

y luego se tiene la línea en alguna parte del programa:

```
if(rele==ON)
```

Cuando el pre-procesador se ejecuta se reemplaza por:

```
if(rele==1;)
```

que genera un error de compilación.

#### 4.4.B-Substitución de macros con argumentos.

También es posible definir macros con argumentos, para que el texto de reemplazo pueda ser diferente para diferentes llamadas de la macro. Por ejemplo:

```
#define MAX(A,B) ((A) > (B)) ? (A) : (B)
```

Aunque aparenta ser una llamada a función, el uso de MAX se expande a código. Cada ocurrencia de un parámetro formal (A o B) será reemplazada por el argumento real correspondiente. Así la línea:

```
x = MAX(p+q,r+s);
```

es reemplazada por la línea:

```
x = ((p+q)>(r+s)) ? (p+q) : (r+s);
```

En tanto que los argumentos se traten consistentemente, esta macro servirá para cualquier tipo de datos; no hay necesidad de diferentes tipos de max para diferentes tipos de datos, como lo habría con las funciones.

Sin embargo, examinando detalladamente la expansión de la macro surgen algunos riesgos a tener en cuenta (expansión: cuando la macro es reemplazada). Las expresiones se evalúan dos veces, veamos el siguiente ejemplo:

```
x = MAX(i++, p++);          /*INCORRECTO*/
```

```
x = ((i++) > (p++)) ? (i++) : (p++);
```

Es incorrecto debido a que se incrementan dos veces las variables, una vez cuando se evalúa cuál es mayor y una segunda vez cuando se hace la asignación a x.

#### 4.4.C-Directivas condicionales.

Mediante las directivas condicionales se busca que se compilen distintas versiones de código mediante el uso de ciertos parámetros.



Una clásica estructura podría ser la siguiente:

**#ifdef** *definicion*

#

#

#

**#elif** *definicion*

#

**#else**

#

**#endif**

Por ejemplo:

```
#if defined(F020)
```

```
    #include<c8051f020.h>
```

```
    #include<def_f020.h>
```

```
#elif F130
```

```
    #include<c8051f130.h>
```

```
    #include<def_f130.h>
```

```
#else
```

```
    #error "Debe definir el uC"
```

```
#endif
```

#### 4.4.D-El operador ##.

El operador **##** del pre-procesador proporciona una forma de concatenar argumentos reales durante la expansión de una macro. Si un parámetro que está en el texto de reemplazo es adyacente a un **##**, es reemplazado por el argumento real, se eliminan el **##** y los espacios en blanco que lo rodean. También proporciona una forma de concatenar texto en forma estática sin que éste sea un argumento.

Por ejemplo:

```
#define JOIN(x,y) x ## y
```

La llamada a JOIN(var,123) produce **var123**. Sin embargo la llamada a JOIN(JOIN(1,2),3) está indefinida: la presencia de **##** impide que los argumentos de la llamada más externa sean expandidos. Así se produce la cadena:

```
JOIN(1,2)3
```

Para solucionar esto se suele introducir un segundo nivel de macro-definición:

```
#define XJOIN(x,y) JOIN(x,y)
```

Ahora sí llamamos a XJOIN(XJOIN(1,2),3) y obtenemos **123**, debido a que XJOIN no involucra al operador ## por lo que no concatena.

La conclusión es que se debe evitar en lo posible el anidamiento de macros.

Ahora veamos un ejemplo en el cual incluimos texto que no estaba en los argumentos; de la misma forma:

```
#define SETpINpORT(PORT,PIN,VALUE) PORT##__##PIN = VALUE
```

Para esto tenemos que previamente tener definida una variable **puerto1\_1**:

```
#include<REG51.h>
```

```
sbit P0_0 = P0^0;
```

#### 4.4.E-Operador \.

Si necesitamos que el reemplazo abarque más de una línea de código, debemos indicarlo con el operador \, de esta forma al juntarse el operador con el caracter de nueva línea este último es ignorado, veamos un ejemplo:

```
#define MicODIGO      \  
{                    \  
unsigned char i;      \  
for (i=0; i<100; i++);  \  
}
```

#### **4.5-Campos de bits y uniones.**

Los campos de bits son estructuras mediante las cuales se puede controlar el tamaño en memoria que se le asigna a una determinada variable. Conceptualmente es un conjunto de bits adyacentes dentro de una unidad de almacenamiento definida por la implantación. Surgió bajo dos premisas, la primera era ocupar menos espacio en la memoria y la segunda proporcionar un acceso más sencillo a las variables que utilizando máscaras. Importante: por más que permite generar variables de 1 bit o más, el espacio que finalmente se ocupará en memoria es el mínimo que asigna el sistema; por lo tanto si trabajamos en un microcontrolador de 8 bits aunque usemos sólo 2 bits nuestra estructura ocupará los 8 bits.

La ventaja de este tipo de estructuras es poder empaquetar datos, por ejemplo toda una trama de transmisión pudiendo acceder a los flags que ésta tenga individualmente.

Por ejemplo:

```
struct PackedData{
    unsigned char bitDeControl: 1;
    unsigned char bitDeACK: 1;
    unsigned char bitsDeSync: 2;
    unsigned char bitsAddress: 4;
    unsigned char byteData1: 8;
    unsigned char byteData2: 8;
    unsigned char byteData3: 8;
};
```

El número que le sigue al carácter dos puntos representa el ancho del campo en bits; los campos son declarados unsigned char para asegurar que sean cantidades sin signo.

Una "union" es una variable que puede contener objetos de diferentes tipos y tamaños, además proveen una forma de manejar distintos tipos de datos dentro de una sola área de memoria de almacenamiento. Continuando con el ejemplo anterior:

```
union myData{
    struct PackedData data;
    unsigned long    var;
};
```

#### **4.6-Typedef.**

La palabra reservada **typedef** nos permite crear nuestros propios tipos de datos. Por ejemplo:

```
typedef unsigned char byteData;
byteData payloadForI2C;
```

#### **4.7-Generando Tablas constantes en memoria de programa.**

Las tablas constantes en memoria de programa no son una herramienta propiamente dicha de C sino en cambio más bien una técnica de programación muy útil en microncontroladores. Básicamente consiste en utilizar memoria de programa para almacenar constantes, por los motivos que se citan a continuación: velocidad de procesamiento de algunas funciones matemáticas, no se cuenta con tanta memoria ram como para desperdiciarla guardando

valores que no van a cambiar sumado al tiempo de inicialización de cada variable, etc.

Por lo tanto se utiliza un vector declarado en memoria de programa al cual se lo puede acceder utilizando un índice, como ventaja se puede ver que se puede ir moviendo el índice secuencialmente (es decir, se cuenta con acceso secuencial), o se puede directamente acceder en forma aleatoria a una determinada posición. Teniendo en cuenta esto, sólo hará falta referenciar el índice con la magnitud de interés a representar.

Por ejemplo: se necesita calcular el  $\sin(x)$ . Utilizar la librería `math.h` de C lleva muchos ciclos de instrucción para un microcontrolador de 8 bits como el que se usa en la materia. Por lo tanto podemos calcular el  $\sin(x)$  para distintos ángulos de interés y guardar los resultados en memoria, ya que los resultados nunca cambiarán. Entonces calculamos de 0 a 90 grados los valores y el índice de nuestro vector serán los ángulos.

```
code float[91] = [ 0, 0.017452406, 0.034899497, 0.052335956
```

Angulo	Sin(x)
0	0
1	0,017452406
2	0,034899497
3	0,052335956
4	0,069756474
5	0,087155743

Nota: No es recomendable utilizar variables tipo `float` en microcontroladores de 8 bits, dependerá la aplicación y la precisión que se necesite. Siempre es preferible realizar un estudio sobre cómo representar la misma información sin introducir un error apreciable y consumiendo menos recursos. Por ejemplo, en el caso del seno se sabe que el valor varia de 0 a 1. Por lo tanto, podríamos descartar la parte entera, debido a que no tiene sentido almacenarla si sólo para un par de valores tiene realmente un significado; luego dependiendo de la precisión requerida podemos representar todos los valores detrás de la coma con bytes.