

Sumario

Capítulo I.....	7
Los PIC's y un poco de su historia.....	7
Como funciona un Microcontrolador PIC.....	8
Arquitectura de los PIC's.....	11
Arquitectura Hardvard:.....	11
Entorno de programación.....	12
Hardware Programador.....	13
Programar PIC con GNU/Linux.....	13
El lenguaje C.....	15
El lenguaje C para Microcontroladores PIC.....	16
Estructura de proyecto en XC8 con MPLAB.....	17
Variables en XC8.....	24
Operadores.....	26
Estructuras de Control.....	27
Directivas del Pre-Procesador.....	30
Control del LCD (HD44780).....	31
Manejo de un LCD con XC8.....	33
Interrupciones en XC8.....	42
Declaración de los registros internos.....	50
Punteros.....	50
Funciones.....	54
Punteros a Funciones.....	56
Estructuras.....	59
Uso del conversor A/D.....	64
Capítulo II.....	71
Memoria EEPROM interna del PIC.....	71
El protocolo I2C.....	76
RTC DS1307 (Real Time Clock).....	83

La dirección del dispositivo.....	87
PCF8574.....	94
Protocolo 1-wire.....	98
Sensor de temperatura 1-wire DS18B20.....	99
Capítulo III.....	112
El Puerto USB.....	112
Token y Packets.....	121
Clase USB CDC con PIC16F1455.....	125
Algunos ejemplos de programas para otros PIC.....	131
Encendiendo un LED con PIC12F683.....	131
Encendiendo un LED con PIC16F84.....	132
Manejando botones y LED's con PIC12F629.....	132
Capítulo IV.....	135
ESP32.....	135
Porque mezclar ESP32 con PIC's?.....	137
Alimentación para el ESP32 Thing.....	138
Algunos pines especiales.....	139
El Internet de las Cosas.....	140
Que es HTML?.....	140
Ejemplos de algunas etiquetas HTML.....	141
Servidores web con electrónica.....	143
Que es Ajax?.....	144
ESP32 con Arduino.....	147
Como programar la placa ESP32 Thing.....	149
Hola Mundo con ESP32.....	150
Voltímetro web con Ajax.....	150
Como funcionan las cosas dentro del PIC.....	152
Como funcionan las cosas en la página web.....	154
Como funcionan las cosas en el servidor web.....	158
Electrónica y conexionado del proyecto.....	165

Lectura de un sensor DS18B20 desde una WEB.....	168
Página web para ver los datos del sensor.....	171
Control de pines con botones web.....	173
Que es un Socket de Red?.....	179
Enviando un contador por un Socket con un PIC.....	180
Socket y datos de un sensor DS18B20.....	186
Que es el protocolo MQTT.....	187
Por qué MQTT.....	188
Como funciona MQTT?.....	188
MQTT con PIC's.....	189
Pasos para subscribir a un tópico.....	190
Otros ejemplos con ESP32 y Arduino IDE.....	198
El Internet de las Cosas.....	198
Como funciona.....	198
Riesgos de IOT.....	198
Web Server con ESP32.....	199
Control HTML de un LED con ESP32	202
Implementado AJAX con ESP32.....	205
Bibliografía.....	209

Los PIC's y un poco de su historia.

Los PIC son una familia de microcontroladores tipo RISC fabricados por Microchip Technology Inc. y derivados del PIC1650, originalmente desarrollado por la división de microelectrónica de General Instrument. El nombre actual no es un acrónimo, en realidad el nombre completo es PICmicro, aunque generalmente se utiliza como **Peripheral Interface Controller** (*Controlador de Interfaz Periférico*).

El PIC original se diseñó para ser usado con la CPU CP16000. CPU que en general tenía prestaciones pobres de entradas y salidas, el PIC de 8 bits se desarrolló en 1975 para mejorar el rendimiento de la CP16000 mejorando las prestaciones de entradas y salidas.

El PIC utilizaba microcódigo simple almacenado en ROM para realizar estas tareas.

Aunque el término no se usaba esos años, se trata de un diseño RISC que ejecuta una instrucción cada 4 ciclos del oscilador.

En 1985 la división de microelectrónica de General Instrument se separa como compañía independiente y cambia el nombre a Microchip Technology.

El PIC, sin embargo, se mejoró con EPROM para conseguir un controlador de canal programable. Hoy en día multitud de PIC's vienen con varios periféricos incluidos (módulos de comunicación serie, UART, núcleos de control de motores, etc.) y con memoria de programa desde 512 a 32000 palabras (una palabra corresponde a una instrucción en lenguaje ensamblador, y puede ser de 12, 14, 16 o 32 bits, dependiendo de la familia específica de PIC).

El PIC usa un juego de instrucciones, cuyo número puede variar desde 35 para PIC de gama baja a 70 para los de gama alta. Las instrucciones se clasifican entre las que realizan operaciones entre el acumulador y una constante, entre el acumulador y una posición de memoria, instrucciones de condicionamiento y de salto/retorno, implementación de interrupciones y una para pasar a modo de bajo consumo llamada sleep.

Microchip proporciona un entorno de desarrollo libre llamado MPLAB que incluye un simulador software y un ensamblador. Otras empresas desarrollan compiladores C y BASIC. Microchip también vende compiladores C para los PIC como el XC8 para todos los micros de 8 bits, XC16 para micros de 16 bits, y XC32 para 32 bits.

Para transferir el código desde una PC al PIC normalmente se usa un dispositivo llamado programador. La mayoría de los PIC's que Microchip distribuye incorporan ICSP (In Circuit Serial Programming, programación serie incorporada) o LVP (Low Voltage Programming, programación a bajo voltaje), lo que permite programar el PIC directamente en el circuito destino.

Para la ICSP se usan los pines RB6 y RB7 (En algunos modelos pueden usarse otros pines como el GP0 y GP1 o el RA0 y RA1) como reloj y datos y el MCLR para activar el modo programación aplicando un voltaje de 13 voltios. Existen muchos programadores de PIC, desde los más simples que dejan al software los detalles de comunicaciones, a los más complejos, que pueden verificar el dispositivo a diversas tensiones de alimentación e implementan en hardware casi todas las funcionalidades. Muchos de estos programadores complejos incluyen ellos mismos PIC pre-programados como interfaz para enviar las órdenes al PIC que se desea programar.

La arquitectura se dirige a la maximización de la velocidad, PIC fue uno de los primeros escalares diseños de CPU, y sigue siendo uno de los más simples y más baratos. La arquitectura de Harvard en el que las instrucciones y los datos provienen de diferentes fuentes-simplifica la sincronización y diseño de microcircuitos.

El conjunto de instrucciones es adecuado para la aplicación de tablas de búsqueda rápida en el espacio de programa. Estas búsquedas tienen una instrucción y dos ciclos de instrucción. Muchas de las funciones se pueden modelar de esta manera.

Como funciona un Microcontrolador PIC.

Para entender un poco como funcionan las cosas dentro de un microcontrolador podríamos comenzar definiendo sus partes.

CPU

Dentro de la CPU está la Unidad de Control (UC) quien se encarga de que todo suceda de acuerdo a lo previsto, la UC es pues quien decide cuando y como las acciones se llevan a cabo.

Se lee una instrucción desde la memoria de programa se la pasa al decodificador de instrucciones y el contador de programa (PC) se incrementa en uno para leer en la siguiente dirección en la memoria de programa.

La Unidad de Aritmética y Lógica (ALU).

También dentro de la CPU, es la encargada de resolver los problemas matemáticos y lógicos, cuando por ejemplo hay que resolver una operación de suma, la UC le pasa a la ALU las variables esta lo resuelve y le entrega los resultados a la UC para que esta disponga de ellos, entonces la UC consulta el programa para saber que debe hacer con el resultado.

I/O Ports.

Controlan los pines que son el vínculo con el exterior, estos pines se pueden configurar como entradas o salidas dependiendo de la necesidad.

Memoria de Programa.

Es la memoria donde reside el código de la aplicación. Si por ejemplo hemos programado una alarma, es en esta memoria donde el código de la alarma reside.

La memoria de programa es la que programamos con nuestro programador, esta memoria es del tipo FLASH.

Memoria Ram.

La memoria RAM es una memoria de lectura y escritura pero temporal es decir los datos en ella se mantienen el tiempo que se haya energía en el sistema, si la memoria se queda sin energía la información se pierde. En un microcontrolador es la memoria destinada a guardar las variables temporales de proceso, variables de programa que solo son importantes en ese momento. Todos los microcontroladores tienen memoria RAM, algunos más otros menos pero todos disponen de esta memoria, no solo es usada por el programa para guardar los datos generados en tiempo de ejecución sino que también los registros propios del microcontrolador radican en RAM, registros de configuración, registros de estado, se puede decir que la CPU se comunica con nosotros a través de estos registros son los llamados SFR (Registro de Funciones Especiales).

Stack de Memoria o Pila.

El stack es memoria RAM donde el controlador guarda datos temporales de uso propio de la CPU, no depende del programador y es manejado por el controlador, tiene en la serie 18 de pic's 31 niveles de profundidad lo que mejora notablemente el anidamiento de interrupciones que se verá mas adelante.

Tradicionalmente la pila o stack, sólo se ha utilizado como un espacio de almacenamiento para las direcciones de retorno de subrutinas o rutinas de interrupción, en donde las operaciones para meter y sacar datos de la pila estan escondidos.

En su mayor parte, los usuarios no tienen acceso directo a la información en la pila.

El microcontrolador PIC18 se aparta un poco de esta tradición. Con el núcleo PIC18 nuevo, los usuarios tienen ahora acceso a la pila y se puede modificar el puntero de pila y pila de datos directamente. Habiendo tales niveles de acceso a la pila permite algunas posibilidades de programación única e interesante.

La memoria EEPROM.

Es una memoria permanente como la ROM pero a diferencia de esta, se puede escribir en ella, se la puede borrar y desde luego leer, no todos los

- RC o INTRC provee una solución de ultra bajo costo para el oscilador
- XT optimizado para osciladores de las frecuencias mas usadas
- HS optimizado para manejar resonadores y cristales de alta frecuencia

XT	Standard frequency crystal oscillator	100kHz - 4MHz
HS	High frequency crystal oscillator	DC - 40MHz
HS+PLL	High frequency crystal with 4x PLL	4MHz - 10MHz
LP	Low frequency crystal oscillator	5kHz - 200kHz
RC	External RC oscillator	DC - 4MHz
RCIO	External RC oscillator, OSC2=RA6	DC - 4MHz
INTRC	Internal RC oscillator	Various
EC	External Clock, OSC2=f _{osc} /4	DC - 40MHz
ECIO	External Clock, OSC2=RA6	DC - 40MHz

Modo Sleep.

El procesador puede ser puesto en modo baja potencia por la ejecución de una instrucción SLEEP

- Oscilador del sistema es detenido
- El estado del procesador es mantenido (Diseño estático)
- El funcionamiento del WDT continua, si esta habilitado
- Mínima absorción de corriente - sobre todo debido a las salidas (0.1 – 2.0µA típicos)

Events that wake processor from sleep	
MCLR	Master Clear Pin Asserted (pulled low)
WDT	Watchdog Timer Timeout
INT	INT Pin Interrupt
TMR1	Timer 1 Interrupt (or also TMR3 on PIC18)
ADC	A/D Conversion Complete Interrupt
CMP	Comparator Output Change Interrupt
CCP	Input Capture Event
PORTB	PORTB Interrupt on Change
SSP	Synchronous Serial Port (I ² C Mode) Start / Stop Bit Detect Interrupt
PSP	Parallel Slave Port Read or Write

Entorno de programación.

Se puede usar tanto el MPLAB X como el clásico MPLAB IDE, sin embargo y a pesar de las mayores exigencias de sistema que tiene el X, este presenta un entorno de trabajo con sustanciales mejoras respecto al clásico IDE de Microchip.

El lenguaje C.

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B. Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, en la actualidad es la principal herramienta de programación para electrónica y el desarrollo de sistemas embebidos con microcontroladores.

Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel trabajando directamente con el Hardware.

La primera estandarización del lenguaje C fue en ANSI o ANSI C y uno de los objetivos de diseño del lenguaje C es que sólo sean necesarias unas pocas instrucciones en lenguaje máquina para traducir cada elemento del lenguaje, sin que haga falta un soporte intenso en tiempo de ejecución.

En consecuencia, el lenguaje C está disponible en un amplio abanico de plataformas (seguramente más que cualquier otro lenguaje). Además, a pesar de su naturaleza de bajo nivel, el lenguaje se desarrolló para incentivar la programación independiente del Hardware.

Este es uno de los puntos más interesantes que tiene este lenguaje para electrónica dado que un programa escrito cumpliendo los estándares ANSI-C puede ser portado a cualquier arquitectura de microcontroladores que lo soporte.

Si bien C se desarrolló originalmente (conjuntamente con el sistema operativo Unix, con el que ha estado asociado mucho tiempo) por programadores para programadores ha alcanzado una popularidad enorme, y se ha usado en contextos muy alejados de la programación de sistemas, para la que se diseñó originalmente razón por la cual se lo ha visto vinculado a electrónica de manera creciente desde hace ya varios años siendo hoy un lenguaje genérico para la programación de microcontroladores.

En 1978, Ritchie y Brian Kernighan publicaron la primera edición de el libro “El lenguaje de programación C”, también conocido como La biblia de C. Este libro fue durante años la especificación informal del lenguaje.

El lenguaje C para Microcontroladores PIC.

Siendo el compilador XC8 una mezcla entre dos grandes compiladores como son *Hi-Tech* y *C18* encontrará de gran ayuda la información contenida en el archivo que hace referencia a las librerías del ya histórico C18 `MPLAB_C18_Libraries_51297f.pdf`.

El lenguaje C se conoce como un lenguaje compilado. Básicamente son dos tipos de lenguajes los que se usan para la programación, los interpretados y compilados. Los interpretados son aquellos que necesitan del código fuente para funcionar (P.ej: Java, Python, MicroPython, etc). Los compilados convierten el código fuente en un archivo objeto y éste en un archivo ejecutable `exe` o `com` para windows, y en un archivo `hex` para el caso de los pic o un `S19` para Motorola.

Podemos decir que el lenguaje C es un lenguaje de nivel medio, ya que combina elementos de lenguaje de alto nivel con la funcionalidad del lenguaje ensamblador.

Es un lenguaje estructurado, ya que permite crear procedimientos en bloques dentro de otros procedimientos. Hay que destacar que el C es un lenguaje portable, ya que permite utilizar el mismo código en diferentes equipos y sistemas incluso en distintos microcontroladores.

Todo programa en C consta de una o más funciones, una de las cuales se llama ***main()***, el programa comienza en la función `main()`, todo código en C tiene esta función que es punto de entrada al programa y desde ella se llaman a otras funciones y resto del programa. Las funciones se delimitan con `{ }` y las líneas de programa se terminan con `;` (*salvo excepciones*). XC8 sigue las normativas del Ansi C.

A la hora de programar es conveniente añadir comentarios (cuantos más mejor) para poder saber que función tiene cada parte del código, en caso de que no lo utilicemos durante algún tiempo. Además facilitaremos el trabajo a otros programadores que puedan utilizar nuestro archivo fuente.

Para poner comentarios en un programa escrito en C usamos los símbolos `/*` y `*/`:

```
/* Este es un ejemplo de comentario */
/* También podemos escribir
   un comentario en varias líneas */
```

El símbolo `/*` se coloca al principio del comentario y el símbolo `*/` al final.

El comentario, contenido entre estos dos símbolos, no será tenido en cuenta por el compilador.

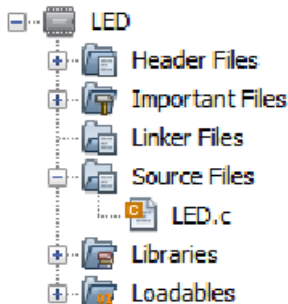
Otra forma de colocar comentarios en algunos compiladores es con el símbolo // solo debemos tener en cuenta que esto solo nos permite una sola línea de comentario.

```
// Este es un ejemplo de comentario
```

Estructura de proyecto en XC8 con MPLAB.

MPLAB trabaja en base a proyectos, en este caso vemos la estructura del proyecto LED.

El árbol de proyecto contiene distintas carpetas y en ellas se irán cargando los archivos que nuestra aplicación necesita.



Todo proyecto tiene un punto de entrada, un archivo donde todo inicia, el archivo principal del proyecto que en este caso se llama **LED.C**. En este archivo se encuentran configuraciones claves para el funcionamiento del controlador como son los fusibles de configuración, el archivo de cabecera del compilador *XC.H*, declaración de variables, etc.

Cuando se crea el proyecto también entre otras cosas se genera la abstracción del Hardware, registros, banderas, mapa de memoria, etc son transparentes al programador (*casi siempre*).

Podemos encontrar líneas como las siguientes:

```
#include <xc.h>
#pragma config
OSC=HS, PWRT=ON, MCLRE=OFF, LVP=OFF, WDT=OFF

#define _XTAL_FREQ 2000000
```

La directiva *#include* significa incluir el archivo *XC.H* los símbolos angulares < > indican que el archivo que estamos incluyendo pertenecen al compilador, podría ser escrito por el usuario en tal caso no se usan símbolos angulares sino “ ” como veremos mas adelante. Todos los archivos *H* son llamados de cabecera (*head*) y son en general archivos que contienen definiciones de variables o colecciones de funciones.

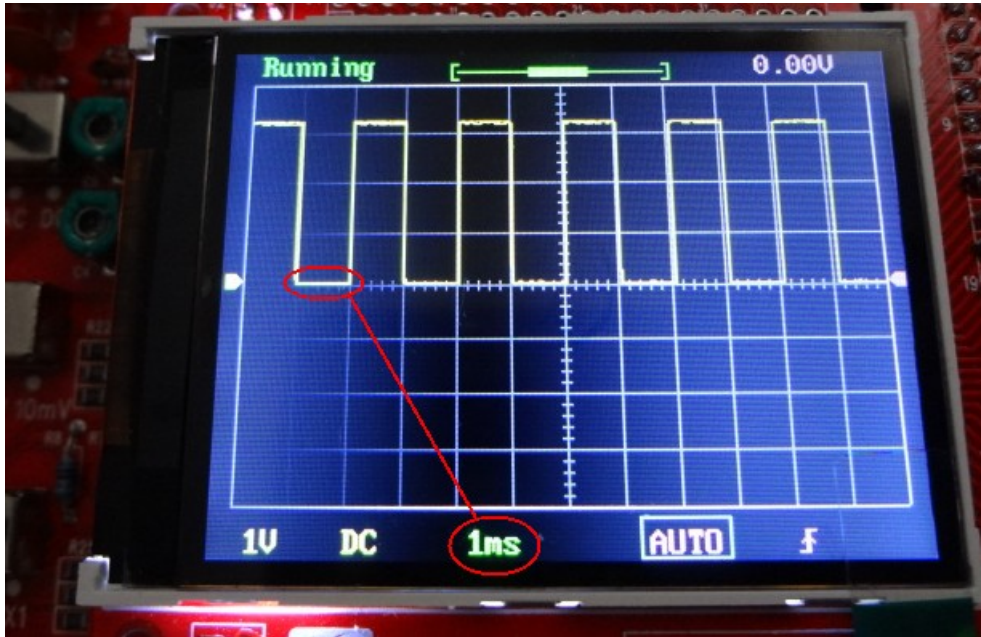
Una función no es otra cosa que un trozo de código que hace algo específico, lo que en ensamblador llamábamos subrutina. En C las funciones están contenidas por llaves de apertura y cierre { }.

```
while (A<B) {
```

Ejecuta esto mientras A sea menor que B.

```
}
```

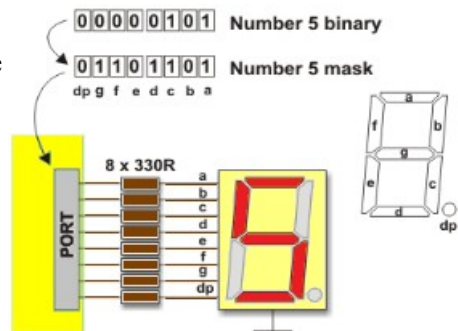
La siguiente imagen es la evolución del pin generando una onda cuadrada con tiempos des 1mS.



Veamos un ejemplo, en este programa vamos a visualizar el estado de un contador a través de un display de siete segmentos tipo cátodo común. El microcontrolador incrementa una variable a velocidad de CPU por lo que es necesario esperar un tiempo entre cada estado de cuenta.

Recuerde, no es prolijo ni recomendable abusar de los *delay()*, son tiempos muertos y mientras el procesador está ocupado en ese retardo no atiende otra cosa.

En nuestros ejemplos usaremos el siguiente formato de conexiones:



El programa inicia con el archivo de cabecera xc8.h, le sigue la configuración de los fusibles o configuración del hard del controlador.

La forma en que el controlador mostrará los datos en el display ha sido “*codificado*” en un array de

16 elementos (un array podemos compararlo con una tabla de elementos del mismo tipo) estos elementos son números que al ser escritos en el puerto B encienden los segmentos (led’s) del display de manera adecuada para cada dígito.

Este arreglo de números (array) como no cambiará con la ejecución del programa podemos guardarla en la *FLASH* y liberar de esta forma memoria *RAM*.



```
const unsigned char Display[16] =  
{0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x67, 0x77,  
0x7C, 0x39, 0x5E, 0x79, 0x71};
```

En XC8 la forma de guardar un array en FLASH se puede ver en esta línea La palabra *const* indica que el array se almacena en memoria **FLASH** y no en **RAM**.

En la segunda línea se lee: unsigned char contador=0. Con esto declaramos una variable tipo *unsigned char* (0-255), esta variable se crea en RAM. Los tipos de variables son como “contenedores” de datos, estos contenedores solo pueden almacenar datos que coincidan con su tipo.(En este simple programa se han omitido las resistencias atenuadoras en los segmentos a fin de simplificar el diagrama electrónico y se a colocado una única resistencia de 270 Ohms).

;;;Atención!!!

Si el string tiene 16 elementos la variable que en si misma puede contener muchos mas elementos dependiendo de su tipo, solo puede direccionar 16 dentro del string, si direcciona mas elementos los resultados son impredecibles!!

En la línea 3 tenemos la función principal *main()*.

La línea 4 configura el puerto B todo como salida (El display se conecta en ese puerto).

En la línea 5 tenemos:

```
PORTB=Display[contador];
```

Esto significa que en puerto B sera igual al contenido del array con el offset de contador.

Veamos un poco mas esto de los array.

Operadores.

Operadores Lógicos

Son los encargados de producir resultados lógicos del tipo TRUE o FALSE.

Operador	Descripción
&&	AND
	OR
!	NOT

Algo a tener en cuenta en el uso de los operandos lógicos es su precedencia, ya que las expresiones vinculadas por && o || son evaluadas de izquierda a derecha, y la evaluación se detiene tan pronto como se conoce el resultado verdadero o falso.

Por ejemplo al realizar:

```
if((k<10) && (++i==100)){  
    }  
}
```

Lo primero que se evalúa es que k sea menor a 10, si no se cumple sin importar el resto salta a la siguiente línea. Si en el diseño del programa se necesitaba el incremento de la variable i el resultado será incorrecto, por lo que debe estudiarse su correcta diagramación.

Operadores de Bit's

Son para modificar los bits de una variable:

Operador	Descripción
&	And
	Or
^	Xor
~	complemento
<<	rotar izquierda
>>	rotar derecha

El operador & es utilizado generalmente para llevar a cero ciertos bits. Por ejemplo:

```
k=0xA5;  
a= 0xF0 & k; // a = 0xA0, k = 0xA5.-
```

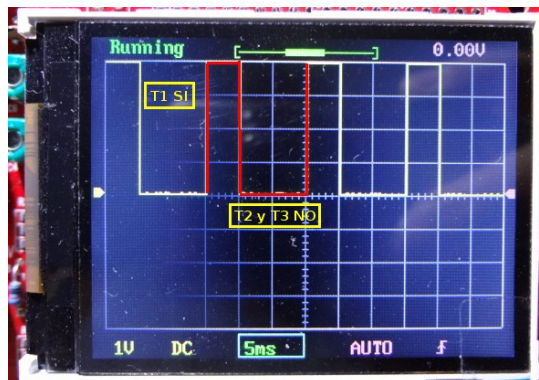
El operador | es utilizado para setear ciertos bits. Por ejemplo:

```
k=0x03;  
a=0x40 | k; // a = 0x43, k = 0x03.-
```

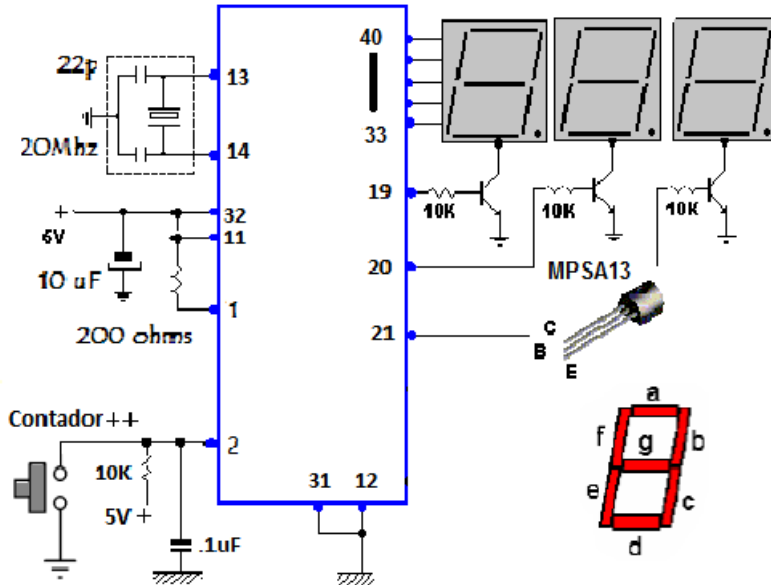
```
}
```

Programa Principal

```
void main(void){ // Inicia la función prncipal
ADCON1 = 0X0F; // Configura los pines del PORT A como I/O
Digitales
TRISA=0X01; // Configura el pin 2 como entrada digital
TRISB=0X00; // Todo el puerto B es puesto a 0
TRISD=0X00; // Configura el puerto D como salida
for(;;){ // Bucle infinito
if(PORTAbits.RA0!=1 && bandera== 0){// Pulsador y bandera =0?
    contador++; // Incrementa el contador
    centena = contador / 100; // Se define la centena
    resto = contador % 100; // Recupera el resto de la div.
    decena = resto /10; // Se define la decena
    unidad = resto % 10; // El resto es la unidad
bandera = 1;
// Evita que se cuente el mismo número mas de una vez.
}
PORTB=Digito[unidad]; // Escribe la unidad en el puerto B
PORTDbits.RD2 = 1; // Pone el pin 21
delay(5); // Espera 5mS
PORTDbits.RD2 = 0; // Pone a 0 el pin 21
PORTB=Digito[decena]; // Escribe la decena en el puerto B
PORTDbits.RD1 = 1; // Pone a 1 el pin 20
delay(5); // Espera 5mS
PORTDbits.RD1 = 0; // Pone a 0 el pin 20
PORTB=Digito[centena]; // Escribe la centena en el puerto B
PORTDbits.RD0 = 1; // Pone a 1 el pin 19
delay(5); // Espera 5mS
PORTDbits.RD0 = 0; // Pone a 0 el pin 19
if(PORTAbits.RA0!=0)
// Espera que se libere el pulsador para contar solo
bandera=0; // un número por vez.
}}
}
```



En la imagen anterior se pueden apreciar los tiempos del multiplexador generando ventanas de conducción para cada transistor de 5 milisegundos y los mismos tiempos de apagado, solo un display está activo mientras que los otros dos permanecen apagados.



Circuito del contador.

Directivas del Pre-Procesador.

El pre-procesador es el primer código que se llama en la etapa de compilación de un programa. El pre-procesador tiene su propio lenguaje y sus directivas inician con un #.

La ventaja que tiene usar el pre-procesador es que los programas son más fáciles de desarrollar, más fáciles de leer y son más sencillos de portar a otros compiladores.

#include

Esta directiva ya la hemos utilizado, se emplea para incluir archivos y suele darse al principio

de los programas, porque en general se desea que su efecto alcance a todo el archivo fuente. Por

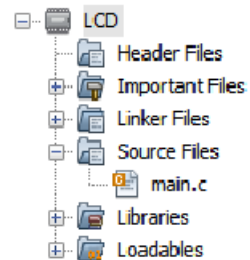
esta razón los archivos preparados para ser incluidos se denominan *headers* o archivos de cabecera.

En ellos se declaran las funciones y definiciones que el programa usará.

#define

Tenemos entonces todas las partes para nuestra aplicación, el árbol del proyecto debe verse así:

```
/* *****  
** File Name      : main.c  
** Author         : Firtec - www.firtec.com.ar  
** Target        : 40PIN PIC18F4620  
** *****/  
#include <xc.h>  
#define _XTAL_FREQ 20000000UL  
#pragma config OSC=HS,PWRT=ON,MCLRE=ON,LVP=OFF,WDT=OFF  
  
#include <stdio.h>
```



```
#include "C:\ELECTRÓNICA\Programas PIC\2014\EJERCICIOS_XC8\LCD.X\lcd_xc8.c"  
#include "C:\ELECTRÓNICA\Programas PIC\2014\EJERCICIOS_XC8\LCD.X\lcd_xc8.h"
```

```
char Buffer[]={"XC8 & Firtec"}; // Cadena de caracteres.  
unsigned int dato_entero= 2014 ;// Valor dato entero  
unsigned int dato_conv= 512 ;  
float voltios =0;  
char str float[7];  
char str int[7];
```

Todos los derechos reservados
© by Firtec Capacitación



```
lcd_puts(str_float);  
lcd_gotoxy(8,2);  
printf(str_int,"E:%04d ", dato_entero);  
lcd_puts(str_int);
```

```
while(1);  
}
```



Resultado al ejecutar el programa.

La clave para la presentación de los datos en el LCD está en la línea:

```
printf(str_float, "F:%2.2f", voltios);
```

Sprintf() es una función de C que se utiliza para darle formato a variables, en este caso estamos tomando el dato *float* contenido en *voltios* y lo convertimos en una cadena de caracteres que lo representará en el LCD. Queremos ver el dato con un formato de dos enteros y dos decimales la letra *f* indica que estamos haciendo referencia a un *float*.

Esta cadena se almacena en *str_float*, incluso guardará la letra *F*:

- *PIE1, PIE2*
- *IPR1, IPR2*

Cada fuente de interrupción tiene tres bits para controlar su operación.

Las funciones de estos bits son:

- Flag para indicar que ha ocurrido una interrupción.
- El bit que permite la ejecución de programa en la dirección del puntero de interrupción cuando se activa el flag.
- El bit de prioridad para seleccionar alta o baja prioridad.

Las características de prioridad de las interrupciones se determinan activando el bit IPEN (RCON<7>).

bit 7 **IPEN:** Interrupt Priority Enable bit
 1 = Enable priority levels on interrupts
 0 = Disable priority levels on interrupts (PIC16XXX Compatibility mode)

RCON: RESET CONTROL REGISTER

R/W-0	R/W-1 ⁽¹⁾	U-0	R/W-1	R-1	R-1	R/W-0 ⁽¹⁾	R/W-0
IPEN	SBOREN	—	RI	TO	PD	POR	BOR
bit 7							bit 0

Cuando se permite la prioridad de las interrupciones, hay dos bits que permiten las interrupciones globalmente. Activando el bit **GIEH** (INTCON<7>) se permiten todas las interrupciones que tengan alta prioridad.

INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF ⁽¹⁾
bit 7							bit 0

Seteando el bit **GIEL** (INTCON<6>) permite todas las interrupciones que tengan baja prioridad.

Cuando la bandera de interrupción esta borrada y la máscara de interrupción este activo para esa interrupción en particular y el bit de las interrupciones globales activo, cuando ocurra una interrupción el programa saltará inmediatamente a la dirección **0x0008** ó **0x0018**, dependiendo del bit de prioridad.

Las interrupciones individuales pueden desactivarse con su correspondiente bit de permiso individual o mascara de interrupción.

Cuando el bit *IPEN* está borrado (*estado por defecto*), las características de prioridad de las interrupciones están desactivadas y las interrupciones son

Los timer cuentan siempre en modo ascendente y parten del módulo hasta el final de cuenta, la interrupción se genera cuando el timer pasa a cero. Un timer en 8 bits podrá contar hasta 255 uno en 16 bits contará hasta 65535.

Veamos el código completo del contador de tres dígitos por interrupción.

```

/*****
** Autor      : Firtec
** Target     : PIC18F4620
** Compilador : XC8
** IDE        : MPLAB X
** XTAL       : 20MHZ
**
*****/
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include <plib/timers.h>

#pragma config OSC=HS,PWRT=ON,MCLRE=OFF,LVP=OFF,WDT=OFF
#ifndef _XTAL_FREQ
#define _XTAL_FREQ 20000000
// Declara el cristal usado.
#endif

void mostrar(void);
// Declara la función que maneja los dígitos.

const unsigned char
Digito[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x7,0x7F,0x67};

unsigned char
contador=0,unidad=0,decena=0,centena=0,resto=0,bandera =0;
volatile unsigned char marca =0; // Variable usada en la
interrupción.

```

RUTINA DE LA INTERRUPCIÓN

```

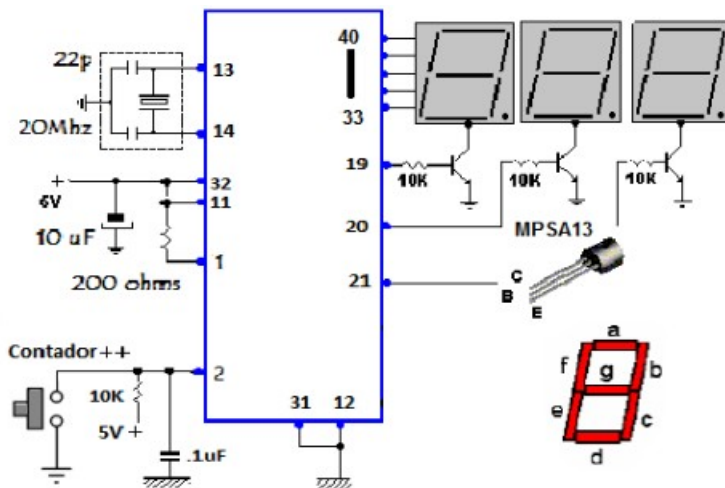
void interrupt high_isr (void){
if (INTCONbits.TMR0IF){ // Verifica bandera TMR0 overflow
WriteTimer0(10); // Configura el módulo del TIM
PORTCbits.RC2 = ~PORTCbits.RC2;
// Cambia de estado pin 33 (LED)
marca++;
// Discrimina cual dígito se activa.
mostrar();
// Función de los dígitos.
INTCONbits.TMR0IF = 0;
// Borra la bandera del TIM
}
}

```

```

break;
}
case 3:{
    PORTDbits.RD1 = 0;    // Pone a 0 el pin 20
    PORTB=Digito[centena];
// Escribe la centena en el puerto B
    PORTDbits.RD0 = 1;    // Pone a 1 el pin 19
break;
}
case 4:{
    PORTDbits.RD0 = 0;    // Pone a 0 el pin 19
    marca =0;
break;
}
}
}
}

```



En el circuito no se ha dibujado el LED en el pin 33 (RC2).

Declaración de los registros internos.

Cada microcontrolador tiene un archivo asociado de cabecera donde se incluyen las declaraciones externas de sus registros internos.

Hay ciertos recursos y posiciones de memoria del controlador que son usados por el compilador y no pueden ser utilizados por los programas de usuario.

Para conocer más sobre esto es recomendado leer la hoja de datos del controlador en uso para conocer su mapa de memoria y registros.

Punteros.

En el libro de Kernighan y Ritchie “*El lenguaje de programación C*”, se define un puntero como «*una variable que contiene la dirección de una variable*».

Tras esta definición clara y precisa, podemos remarcar que un puntero es un tipo especial de variable que almacena el valor de una dirección de memoria.

se repetirá e imprimirá toda la cadena hasta que se encuentre el carácter de fin de cadena 0 ó \0.

Algunos detalles de los punteros en XC8.

- `&` *Asocia la dirección al puntero.*
- `*` *Permite el acceso al contenido.*
- `char *ptr;` *Declaración de una variable puntero a una variable char.*
- `int *ptr;` *Declaración de una variable puntero a una variable int.*
- `unsigned char *ptr;` *Declaración de un puntero a un unsigned char.*
- `Data1 = *ptr + 10;` *A lo que apunta ptr se le suma 10 y se guarda en Data1.*
- `(*ptr)++;` *Se incrementa en 1 lo que apunta ptr.*
- `*ptr++;` *Se incrementa el puntero a la siguiente dirección de variable*
- `++*ptr;` *Se incrementa en 1 lo que apunta ptr.*
- `ptr2=ptr;` *Se asigna al segundo puntero lo que apunta ptr.*
Ósea los 2 punteros apuntan a la misma variable.
- `rom unsigned char *ptr_rom;` *Puntero a memoria Flash.*
- `ram char *ptr_ram;` *Puntero a memoria RAM*

Un ejemplo de un puntero a un array.

```
#pragma romdata seccionDisplay
const rom unsigned char Digito[10] = {1,2,3,4,5,6,7,8,9};
// Array FLASH
#pragma romdata
char Buffer[17]={"Firtec Rosario!!"}; // Array RAM
unsigned char *ptr_rom; // Puntero a memoria Flash
char *ptr_ram; // Puntero a memoria RAM
.
.
void main(void){
lcd_init();
ptr_ram = & Buffer[0]; // El puntero apunta al primer
elemento del array
ptr_rom = &Digito[1]; // Puntero apunta al segundo
elemento de array
lcd_gotoxy(1,1);
```

```

voltage = R * I; /* Ley de Ohm */
return voltage; /* Retorna el resultado */
}

```

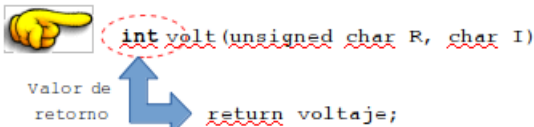
La estructura general del encabezado de una función es:

<u>VariableDeRetorno</u>	<u>NombreDeFunción</u>	<u>(ListaDeArgumentos)</u>
↓	↓	↓
<u>int</u>	<u>volt</u>	<u>(unsigned char R, char I)</u>

Los argumentos se encuentran entre paréntesis, existen dos tipos de paso de argumentos, por valor y por referencia. Los utilizados en la función anterior son del tipo por valor, pues no alteran directamente el contenido de la variable, sino que se realiza una copia de la variable sin alterar el contenido.

El cuerpo realiza la labor específica de la función, en este caso, calcula el voltaje. Inicialmente se declara una variable temporal `voltage`, las variables declaradas internamente en la función, solo existen dentro de ella, cuando la función cumple su cometido (regresando a la función principal) la variable `voltage` desaparecerá del sistema.

Este concepto sobre funciones y variables es muy importante porque una variable declarada como global será accesible desde cualquier parte del programa sin embargo variables locales a las funciones solo son accesibles dentro de la función, las variables locales existen solo cuando son invocadas en la función, las variables globales están siempre en memoria disponibles en cualquier momento y por cualquier parte del programa.



En el encabezado de la función se especifica la variable que se va a retornar, volviendo a la función `volt()`, esta retorna un tipo entero.

Si observamos más a fondo el cuerpo de la función, la primera línea de la izquierda nos describe que variable va a ser retornada en el caso que tuviera retorno, una función que no retorna valor y no recibe argumentos sería:

```
void Nombre_Función(void);
```

Esta variable (`voltage`) también fue declarada tipo entero, sino el compilador dará advertencias parciales sobre la variable a retornar. No necesariamente tuviésemos que haber escrito la función con una variable temporal de cálculo, la misma acción pudo haberse escrito de la siguiente manera:

```

int volt(unsigned char R, char I) {
    return (R*I);
}

```

La función tiene el mismo fin o propósito que la original y dará el mismo resultado.

La función debe ser llamada especificando el mismo tipo de variable como argumentos a utilizar.

Observe esta parte del código:

```

unsigned char a;
char b;
int resultado;
a = 2;
b = 5;
resultado = volt(a,b); /* Utilizando variables
*/
resultado = volt(9,-1); /* Utilizando constantes
*/

```

Preferiblemente se prefiere escribir la declaración de la función en los archivos *.h y la definición en los archivos *.c, de tal manera que solo llamáramos la librería correspondiente a todas las funciones y utilizáramos solo las que necesitamos.

Imaginemos que existe una librería de funciones eléctricas (FE.h) que llama a la función volt() para calcular el voltaje y utilicemos la librería LCD.h para imprimir en pantalla el voltaje. Ambas librerías creadas por el programador en otro momento y que contiene en su interior todas las funciones necesarias para realizar este tipo de tareas.

Es importante entender la diferencia entre declarar una variable dentro de una función o declararla global, fuera de la función `main()`, si la variable es global se usará un vector de RAM para contener el dato sin embargo si la variable es local a la función el compilador usará algún registro de CPU disponible o incluso el propio acumulador. Declarar variables locales ahorra RAM y es una práctica muy aconsejable sabiendo siempre del alcance local de la variable.

Punteros a Funciones.

Las funciones y los punteros pueden trabajar juntos, podemos usar un puntero para llamar a funciones o incluso pasar variables a funciones apuntadas por punteros.

En el siguiente ejemplo hay un puntero *fp* que se vincula distintas funciones según el caso.

En este ejemplo el puntero apuntara a distintas funciones pero estas trabajan con datos del tipo float por lo tanto el puntero deberá tener esta naturaleza.

```
float (*fp)(float)
```

En el ejemplo propuesto el puntero apunta dos funciones que escala los datos de distintos canales analógicos. El ejemplo si bien es operativo y funcional solo pretende ser demostrativo seguramente usted encontrará aplicaciones mas interesantes para su uso.

```
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include <plib/adc.h> // Librería para el ADC
#pragma config OSC=HS, PWRT=ON, MCLRE=OFF, LVP=OFF, WDT=OFF
#ifndef _XTAL_FREQ
#define _XTAL_FREQ 20000000
#endif
#include "C:\ELECTRÓNICA\EJERCICIOS_XC8\Puntero_Funcion.X\
lcd_xc8.c"
#include "C:\ELECTRONICA\LCD.X\lcd_xc8.h"

void Mide_Conversor(void);
void Config_ADC(void);
void STR_AN0(float dato);
void STR_AN1(float dato);

char Str_AN0[5]; // String de 4 elementos
char Str_AN1[5];
float voltaje = 0.00;
unsigned char conversiones=0;
unsigned int M0=0;
volatile unsigned char bandera =0;
unsigned char canal =0;

float (*fp)(float) = NULL; // Se declara un puntero a función
```

Se discrimina el origen de la interrupción

```
void interrupt high_isr (void){
    if(PIR1bits.ADIF) // Verifica la bandera de inte. del
    conversor A/D
        bandera++;
    PIR1bits.ADIF=0; // Borra la bandera de Interrupcion del
    conversor A/D
}
```

Se recomienda que la máxima resistencia de entrada (R_s) sea de 2.5K, sino la conversión no sería del todo exacta en mediciones de señales dinámicas, por lo que habría que hacer una adaptación de impedancias entre las partes, sin embargo para mediciones estáticas o que varían muy lentamente esto es despreciable.

(Esto último para la serie 18 de pic's. En la serie 16 la R de entrada es de 10K)

Tiempo de Adquisición.

Para que el conversor posea la exactitud especificada, se debe contemplar la carga del condensador CHOLD de manera que entre el nivel de tensión del canal. La impedancia de la fuente (R_S) y la impedancia del cambio del muestreo interno (R_{SS}) afectan directamente al tiempo requerido para cargar el condensador CHOLD. La impedancia recomendada máxima para las fuentes analógicas es de 2,5k Ω . Cuando la impedancia decrece, el tiempo de adquisición también decrece. Después de que el canal analógico haya sido seleccionado este tiempo de adquisición debe tenerse en cuenta antes de arrancar la conversión.

El calculo detallado puede observarse en el datasheet del PICs utilizado, a modo de ejemplo vemos que en un 16F87XA el mínimo tiempo de adquisición es de 19.72us el cual debe ser generado por software cada vez que se seleccione un canal.

En cambio para los PIC18Fxx el tiempo mínimo de adquisición se reduce a 2.45us.

El tiempo por bit de la conversión A/D se define como TAD. La conversión A/D necesita un mínimo de 12TAD para la serie 16F o 11TAD para la serie 18F con 10 bits de conversión.

La fuente de reloj A/D de la conversión es seleccionable por software. Hay siete opciones posibles.

- 2 TOSC.
- 4 TOSC.
- 8 TOSC.
- 16 TOSC.
- 32 TOSC.
- 64 TOSC.
- Oscilador interno RC

Para el TAD las opciones posibles son:

- 20 x TAD.
- 16 x TAD.
- 12 x TAD.

- $8 \times TAD$.
- $6 \times TAD$.
- $4 \times TAD$.
- $2 \times TAD$.
- $0 \times TAD$.

Para las conversiones A/D correctas, el reloj de conversión A/D (TAD) debe ser tan corto como sea posible pero mayor que un mínimo TAD (para más información véase datasheet del PIC utilizado) por ejemplo para la serie 16F en 1.6us y para la serie 18F en 0.7us.

Salir fuera de los rangos operativos puede terminar en un daño permanente del conversor.

Veamos un ejemplo:

Trabajando a 20Mhz. $1/20 = 50nS$

$32 \times 50nS = 1600nS$ (1,6uS) El limite a respetar es **0,7uS**.

No podemos tener un tiempo menor a 700nS.

El tiempo mínimo de adquisición no puede ser menor a **2,45uS**.

$$2 \times 1,6uS = 3,2uS$$

También podríamos haber usados los siguientes valores:

$$16 \times 50nS = 800nS \quad 4 \times 0,8uS = 3,2uS$$

Recordar:

- **Tosc** Periodo del circuito de oscilacion (en nuestro caso el cristal).
- **Tacqt** Significa Periodo de Adquisicion, es el tiempo necesario para que el circuito sample/hold guarde la muestra antes de ser convertida, es decir el tiempo que el capacitor de mustreo se cargue con el voltaje a convertir. Esto se hace para evitar errores en la conversion.
- **Tad** Significa Periodo de conversion analogo digital, es el tiempo en que el ADC realiza la conversion de cada bit.

Algunos ejemplos de selección del Tad.

Si uso $96MHz/2=48MHz$

$$Tad=64 \cdot Tosc=64/Fosc=64/48MHz=1.3333us$$

Utilizo $96MHz/4=24MHz$ porque

$$Tad=32 \cdot Tosc=32/Fosc=32/24MHz=1.3333us$$

Utilizo $96MHz/6=16MHz$ porque

$$Tad=16 \cdot Tosc=16/Fosc=16/16MHz=1us$$

Pero el tiempo de cada ciclo aumenta lo que disminuye el tiempo total de conversión

Si uso un cristal de 20MHz,

$$Tad=16 \cdot Tosc=16/Fosc=16/20MHz=800ns$$

Si uso el oscilador interno a 8MHz,

```

Kbhit=1;           // Indica que se ha recibido un dato
PIR1bits.RCIF=0; // Borra bandera de interrupción
}
}

```

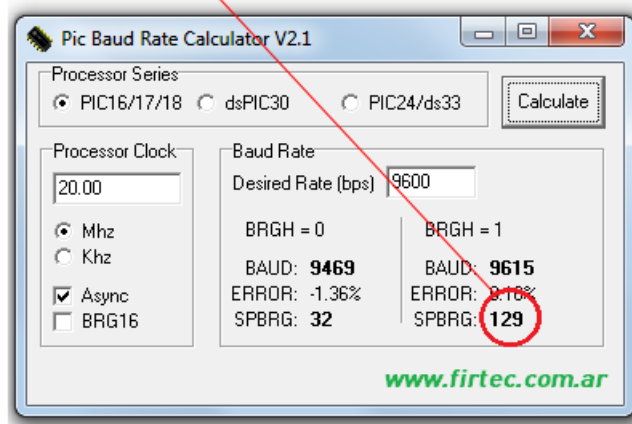
Función Principal

```

void main(void){
unsigned char base =1;

// Inicializa el USART y lo configura a 8N1, 9600 baud
OpenUSART (USART_TX_INT_OFF & // TX sin interrupción
USART_RX_INT_ON & // RX con interrupciones
USART_ASYNCH_MODE & // Modo asincrónico
USART_EIGHT_BIT & // Modo alta velocidad
USART_CONT_RX & // Recepción continua
USART_BRGH_HIGH, 129); // 9600 baudios a 20Mhz

```



Para configurar los baudios podemos usar la herramienta Pic Baud Rate

```

RCONbits.IPEN = 0; // Deshabilitamos prioridades
INTCONbits.PEIE=1;
// Habilitamos la interrupción de perifericos
lcd_init();
//Delay1KTCYx(25);
lcd_puts("USART PIC18F4620"); // Cartel inicial
INTCONbits.GIE=1; //Habilita interrupción global
while(1){
if(Kbhit !=0){ // Indica que hay nuevos datos recibidos
Kbhit = 0; // Borra la marca de ISR
if(Data!=0x00){ // El dato es distinto de 0?
if(Data==0x08 & base > 0){ // Backspace & base son verdaderos?
base--;
lcd_gotoxy(base,2);
lcd_puts(" ");
}
else{
lcd_gotoxy(base,2);
lcd_putc(Data); // Muestra el dato en pantalla
}
}
}
}

```

Se recomienda leer la hoja de datos de la memoria para mayor información sobre señales y tiempos de acceso.

Memoria 24LC256 con XC8.

```

/* *****
** File Name      : 24LC256.c
** Version       : 1.0
** Description    : Control de una memoria 24LC256
**               : Memoria de 32Kx8=256Kbit 64Bytes x Paginas
** Target        : 40PIN PIC18F4620
** Compiler      : Microchip XC8
** IDE           : Microchip MPLAB IDE v8.50
** XTAL          : 20MHZ
** *****/
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include <plib/i2c.h>

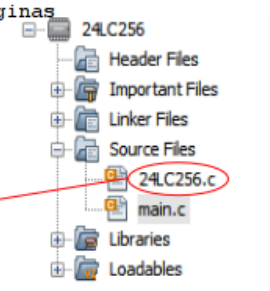
#include "24LC256.h"

#pragma config OSC=HS,PWRT=ON,MCLRE=OFF,LVP=OFF,WDT=OFF

#define _XTAL_FREQ 20000000

#include "C:\ELECTRÓNICA\Programas PIC\2014\
EJERCICIOS_XC8\24LC256.X\lcd_xc8.c"
#include "C:\ELECTRÓNICA\Programas PIC\2014\
EJERCICIOS_XC8\24LC256.X\lcd_xc8.h"

```



FUNCIÓN PRINCIPAL DEL PROGRAMA

```

void main(void) {
unsigned char dir_hi =0, dir_low= 500;
unsigned char Dato=54;          // Dato cualquiera a guardar en
Memoria
char str_int[14];
  lcd_init();
  OpenI2C(MASTER,SLEW_OFF);    // Modo Master
  SSPADD = 49;                 // 100KHz para 20MHz
  Escribe_Byte(0xA0,dir_hi,dir_low,Dato);
// Escribe la memoria

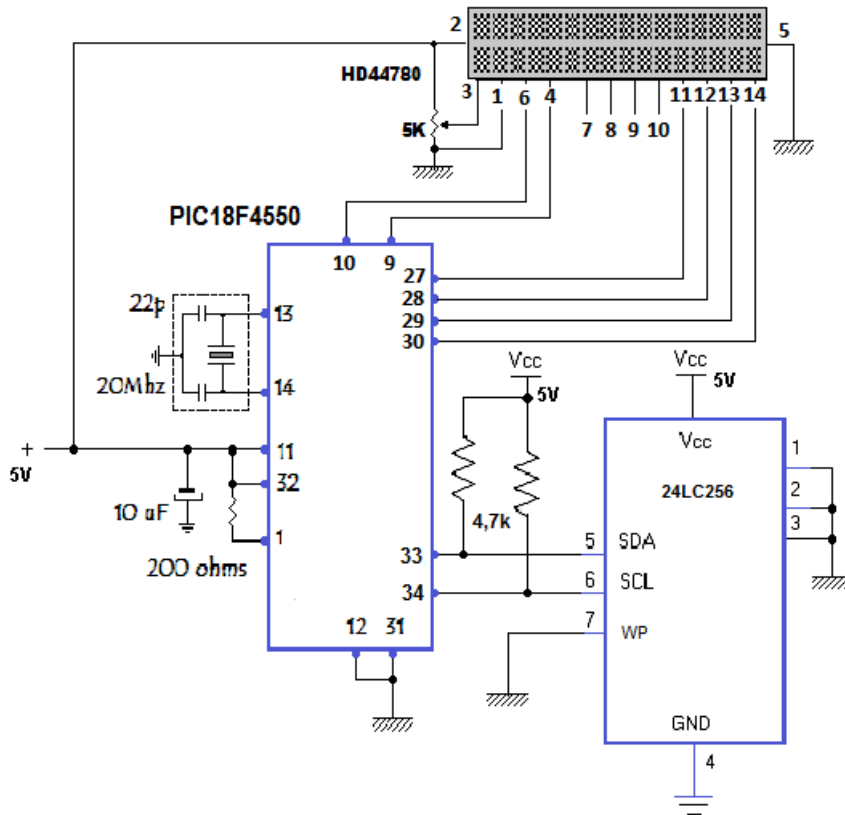
  lcd_puts(" Memoria 24C256");
  lcd_gotoxy(1,2);
  sprintf(str_int,"Dato memoria:%02d
",Lee_Byte(0xA0,dir_hi,dir_low));
  lcd_puts(str_int);
while(1);
}

```

Para el funcionamiento de este ejemplo necesitamos de dos archivos encargados de lidiar con las funciones relativas a la memoria.

- *24LC256* Contiene las funciones de control de la memoria.
- *24LC256.H* Contiene las declaraciones de estas funciones.

Todo lo referente al propio bus *I2C* esta contenido en el archivo de cabecera *i2c.h* provisto por el compilador. El siguiente es el circuito del ejemplo propuesto.



El programa ejemplo solo escribe un byte en la posición 500 y lo recupera para mostrarlo en LCD.

Librerías para el uso de las memorias EEPROM externas.

```

/*
* File:    24LC256.c
* Author:  Firtec
* www.firtec.com.ar - informes@firtec.com.ar
* Created on 24 de mayo de 2014, 10:42

```

```

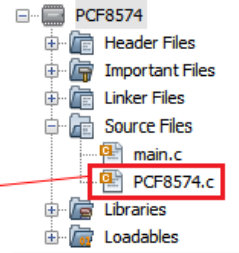
/*****
* File:   PCF8574.c
* Author: Firtec
* www.firtec.com.ar - informes@firtec.com.ar
* Created on 24 de mayo de 2014
*****/

#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include <plib/i2c.h>
#include "PCF8574.h"

#pragma config OSC=HS, PWRT=ON, MCLRE=OFF, LVP=OFF, WDT=OFF

#define _XTAL_FREQ 20000000

```



FUNCIÓN PRINCIPAL DEL PROGRAMA

```

void main(void) {
    unsigned char byte = 0;
    PORTB=0;
    TRISCbits.TRISC4=1;
    TRISB = 0;
    ADCON1=0x0F; // No hay canales analógicos
    OpenI2C(MASTER, SLEW_OFF); // Master, 100KHz
    SSPADD = 49;

```

($Fos/Fck*4$)-1 donde Fos 20Mhz (20000Kc) y Fck 100Kz
 (20000Kc/400kc)-1 = 49 para 20Mhz y Ck I2c 100K

BUCLE INFINITO

```

while(1) {
    byte = ByteReadI2C(chip); // Lee el puerto del PCF8574
    byte = ((byte >> 4) & 0x0f) | ((byte << 4) & 0xf0);
    // Swap nibles
    __delay_us(3000);
    ByteWriteI2C(chip,byte); // Escribe el puerto del PCF8574
}
}

```

Este simple programa lee cuatro bits del puerto del *PCF8574* intercambia los 4 bits de menor peso con los de mayor peso y los envía nuevamente al puerto. Este simple driver contiene el código necesario para acceder al dispositivo.

```

/*****
* File:   PCF8574.c
* Author: Firtec
* www.firtec.com.ar - informes@firtec.com.ar
* Created on 24 de mayo de 2019, 10:42
*****/

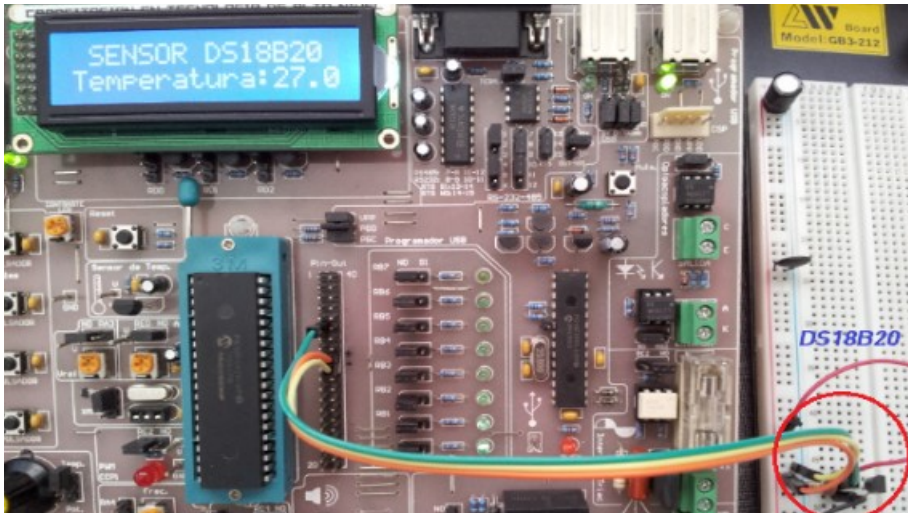
```

```

float Leer_DS18B20(void)
{
    unsigned char temp1, temp2;
    int temp3;
    float result;
    Reset_1wire();
    Escribir_Byte(0xCC);
    Escribir_Byte(0x44);

    Reset_1wire();
    Escribir_Byte(0xCC);
    Escribir_Byte(0xBE);
    temp1 = Leer_Byte();
    temp2 = Leer_Byte();
    temp3 = (( int) ((( int ) ( temp2 ) ) << 8 ) + ((int )
( temp1 ))) );
    result = (float) temp3 / 16.0;
    return(result);
}

```



Programa en ejecución.

En ocasiones puede ser necesario necesitar leer el código ROM de los sensores.

En el ejemplo siguiente vemos la forma de leer este código y mostrarlo en un LCD. Este número de 64 bits puede ser útil para realizar una red de sensores e interrogarlos por su número ROM.

Podríamos tener varios sensores y tratarlos por su número de “RED”. Ejemplo para leer el número de serie de cada sensor.

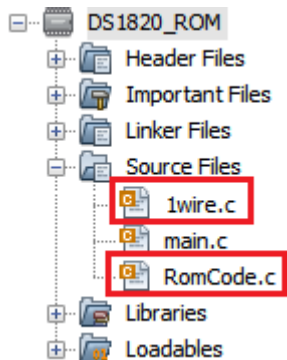
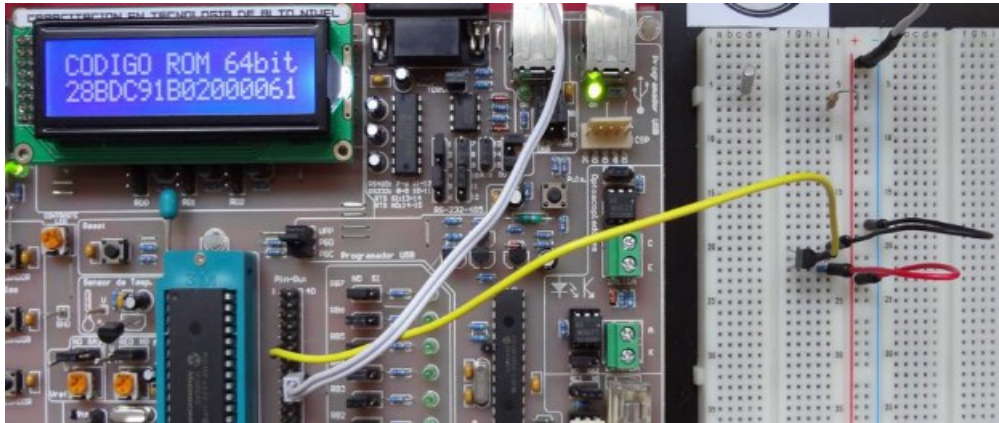
```

}
lcd_gotoxy(1,2);
lcd_puts("                "); // Limpia pantalla
LEER_ROM(); // Se leen los 64 bit's del sensor
lcd_gotoxy(1,2);
for(puntero_array=0;puntero_array<8;puntero_array++){
    sprintf(a,"%02X",ROM_DATA[puntero_array]);
    lcd_puts(a);
}

    while(1);

}

```



Se puede apreciar el resultado del ejemplo propuesto. El código siguiente son las librerías necesarias para su funcionamiento.

```

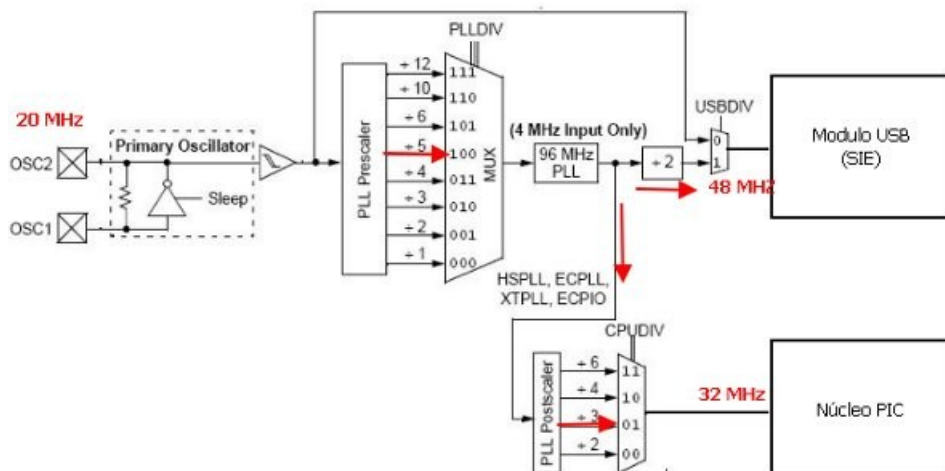
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include "1wire.h"
#include "RomCode.h"
extern unsigned char ROM_DATA [8];
extern unsigned char puntero_array;

```

```

/***** ENVIAR_ROM_CODE *****/
/* Esta función envía los 64 bit's del ROM CODE */
/* Argumentos: Ninguno */
/* Retorna: Nada */
/*****/
void ENVIAR_ROM_CODE(void) {
OWReset ();
    OWWriteByte(0x55);
puntero_array=0;

```



Debido a la complejidad de la comunicación USB lo que ha hecho tanto Microchip como otros desarrolladores de compiladores es proporcionar librerías que comúnmente se les llama "Stacks" o pila de software cuya finalidad es facilitar la tarea al programador de dispositivos, de tal forma que no es necesario conocer a fondo el protocolo de comunicación USB, simplemente se necesita saber que funciones públicas me proporciona el "Stack" correspondiente para poder enviar y recibir los datos a través del bus. Para poder hacer uso del Stack USB desde C18 debemos tener instalado *Microchip Application Libraries* que se descarga de manera libre desde el sitio de Microchip y que se incluye en el DVD del presente libro. Una vez que instalamos la aplicación tendremos agregados no solo el Stack USB sino también el Stack TCPIP, Smart Card, etc.

Comunicación USB.

Como se comentó antes, la comunicación USB entre el host y el dispositivo está basada en pipes o canales lógicos. Los pipes son conexiones del host a una entidad lógica, se realiza través de los endpoint o punto final. No son conexiones físicas, sino que son establecidas por el host en la configuración inicial, para el manejo del protocolo.

En un dispositivo full speed se establecen 32 pipes lógicas, entre el software de la aplicación y el interface del dispositivo. Existe un pipe por defecto (**End Point 0**) que se utiliza en las transferencias de configuración del dispositivo, como por ejemplo la enumeración.

Se establece una conexión lógica entre la aplicación cliente y los diferentes interfaces del dispositivo. Cada interface no es más que el agrupamiento de los

Cuando un dispositivo se excede en el consumo, se apaga, cortándole el suministro de corriente, de forma que no afecte al funcionamiento del resto de los dispositivos conectados al puerto.

El estándar exige que los periféricos conectados lo hagan en un modo de bajo consumo, de 100mA, y luego le comuniquen al host cuanta corriente precisan. Posteriormente pueden cambiar a un modo de alto consumo, si se lo permite el host. Los dispositivos que no cumplan con los requisitos de potencia y consuman más corriente de la negociada con el host pueden dejar de funcionar sin previo aviso o en algún caso dejar el bus sin funcionar. Es por esto que se debe tener cuidado a la hora de alimentar prototipos o proyectos con los 5 voltios provistos por el puerto USB.

Clase USB CDC con PIC16F1455.

Se puede ver aquí un ejemplo de como implementar las comunicaciones con la clase CDC. (*Communications Device Class*).

En general para los proyectos que incorporan múltiples archivos podríamos decir que no importa conocer al detalle el contenido de cada archivo pero si saber cuales hay que agregar al proyecto, las librerías están para ser usadas casi de modo abstracto.

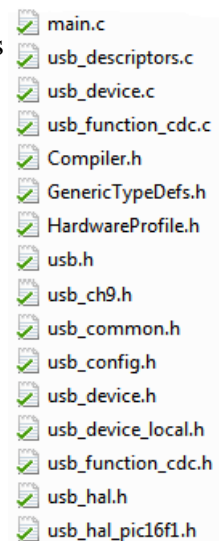
Sin embargo hay dos archivos que si podemos alterar, ellos son *HardwareProfile.h* y *usb_config.h*.

El primero para definir el hardware y el segundo para determinar el comportamiento del dispositivo USB.

Entre las características más importantes de la clase CDC destacan las siguientes:

- Tasa de transferencia máxima de 80 Kbytes/s.
- Las librerías compiladas ocupan un tamaño reducido (4Kb).
- Resuelve toda la comunicación en software, sin requerir hardware adicional.
- El flujo de datos es manejado enteramente por el protocolo

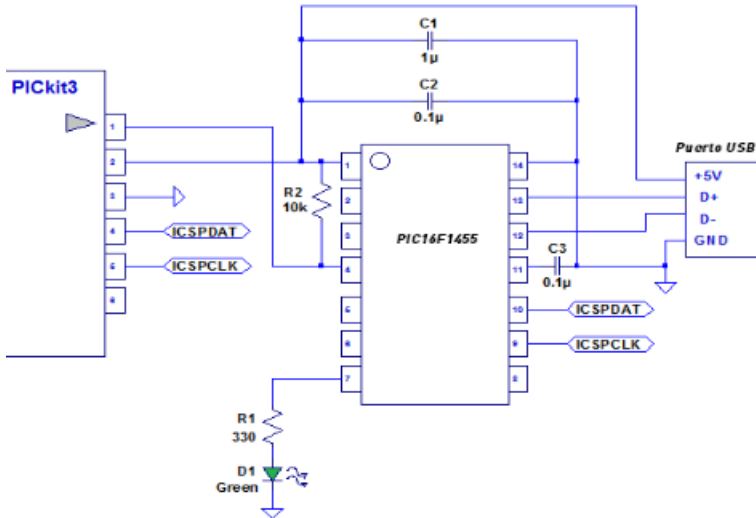
USB (no es necesario usar XON/XOFF ni control de flujo por hardware).



En la función *main()* un bucle infinito *while()* maneja las tareas USB.

```
while (1)
{
    #if defined(USB_POLLING)
        USBDeviceTasks();
    #endif
}
```

Una vez que la instalación finalice, y cada vez que se conecte el dispositivo se creará un puerto COM virtual sobre el USB.



Circuito propuesto para el ejemplo.

Algunos ejemplos de programas para otros PIC. Encendiendo un LED con PIC12F683.

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h> // Librería XC8
#define _XTAL_FREQ 8000000 // Frecuencia de reloj
#define led1 GPIO,GP2

#pragma config
FOSC=INTOSCIO,PWRTE=ON,MCLRE=OFF,BOREN=ON,CPD=OFF,CP=OFF,WDTE=OFF
```

Función principal del programa.

```
void main ()
{
    OSCCONbits.IRCF = 0b111;
/*      111 configura osc. a 8MHz      */
```

```

    CMCON0bits.CM = 0b111;      /* desactiva comparadores */
    ANSELbits.ANS = 0b0000;
/* configura puerto como E/S digitales */
    // TRISIObits.TRISIO0 = 0b000000;
/* configura GP0 como salida */
    TRISIO = 0b000000000; // Configuro puerto B como salidas
while (1) // Bucle infinito
    {
        led1 = APAGADO; // Apago pin RB0
        __delay_ms(100);
        led1 = ENCENDIDO; // Enciendo pin RB0
        __delay_ms(100);
    }
}

```

Encendiendo un LED con PIC16F84.

```

#include <stdio.h>
#include <stdlib.h>
#include <xc.h>

#define _XTAL_FREQ 4000000 // Frecuencia de reloj

// Configuración del controlador
#pragma config FOSC = INTOSCIO
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = OFF
#pragma config BOREN = ON
#pragma config LVP = OFF
#pragma config CPD = OFF
#pragma config CP = OFF

```

FUNCIÓN PRINCIPAL DEL PROGRAMA

```

void main (){
    TRISB = 0b00000000; // Configura puerto B como salidas
while (1) // Bucle infinito
    {
        PORTBbits.RB0 = 0; // Apaga pin RB0
        __delay_ms(500);
        PORTBbits.RB0 = 1; // Enciende pin RB0
        __delay_ms(500);
    }
}

```

Manejando botones y LED's con PIC12F629

```

*****
*
*
*

```

```

*   Asignación de Pines:
*
*       GP1 = Led que indica botón presionado
*
*       GP2 = LED
*
*       GP3 = Botón activo con nivel bajo
*
*
*
*****/

#include <stdio.h>
#include <stdlib.h>
#include <xc.h> // Librería XC8

#define _XTAL_FREQ 8000000 // Frecuencia de reloj

#pragma config
FOSC=INTOSCIO,PWRTE=ON,MCLRE=OFF,BOREN=ON,CPD=OFF,CP=OFF,WDTE=OFF

// Pines del micro.
#define START    GPIObits.GP2           // LED's
#define SUCCESS  GPIObits.GP1

#define BUTTON   GPIObits.GP3           // Pulsador
#define MAXRT    200                    // Maximum reaction time in ms

```

FUNCIÓN PRINCIPAL DEL PROGRAMA

```

void main()
{
    unsigned char    cnt_8ms;           // Variable contador
    TRISIO = 0b111001; // Configura GP1 y GP2 como salidas

    // Configura Timer0
    OPTION_REGbits.T0CS = 0; // Selecciona modo de trabajo
    OPTION_REGbits.PSA = 0; // Pre-escalador asignado al Timer0
    OPTION_REGbits.PS = 0b100; // Prescalador = 32
                                // Incrementa cada 32 us

```

BUCLE PRINCIPAL

```

for (;;)
{
    GPIO = 0; // LED apagado
    __delay_ms(2000); // Delay 2000 ms
    START = 1; // LED de inicio activo

```

- *WiFi 802.11.*
- *Modulo Bluetooth*
- *Opera en un rango de voltajes que va desde 2.2 a 3.6V*
- *28 Pines GPIO*
- *10 Pines con sensibilidad capacitiva para sistemas Touch.*
- *Acelerador de Hardware.*
- *Memoria SPI externa del tipo Flash con 4MB (Aquí se guardan las aplicaciones).*
- *Integra un cargador de baterías.*

Procesador Dual-core Tensilica LX6.

Porque mezclar ESP32 con PIC's?

La respuesta podría ser simplemente “*Y porqué no*”.

Es barato, fácil de conseguir, fácil de usar y se puede conectar con cualquier PIC y brindar conectividad TCP-IP.

Que hay *PIC's* que tiene conectividad TCP-IP es cosa sabida y no se pretende entrar en una competencia tecnológica, la idea es conecta a Internet un desarrollo con un simple PIC12Fxx o un PIC16Fxx y no tener que portar todo el código a un micro mas actual.

Para eso sirve ESP32, permite conectar todo con todo. Imagine enviar datos con un PIC clásico a un protocolo como MQTT, tendríamos un PIC (Sin importar de que tipo) conectado a un protocolo hecho para el Internet de las Cosas.

Estamos comentando ESP32 porque su programación sigue siendo C pero también podría ser Pico W, mas económico pero su programación ya no sería C y estaríamos hablando de MicroPython.

ESP32 incorpora la mayoría de los módulos clásicos que se encuentran en los microcontroladores, conversor, comunicaciones y pines GPIO configurables. Sin embargo y a no ser que nuestra aplicación sea de extrema simpleza, el verdadero poder de ESP32 está en la forma eficiente que logra conectividad WiFi y Bluetooth.

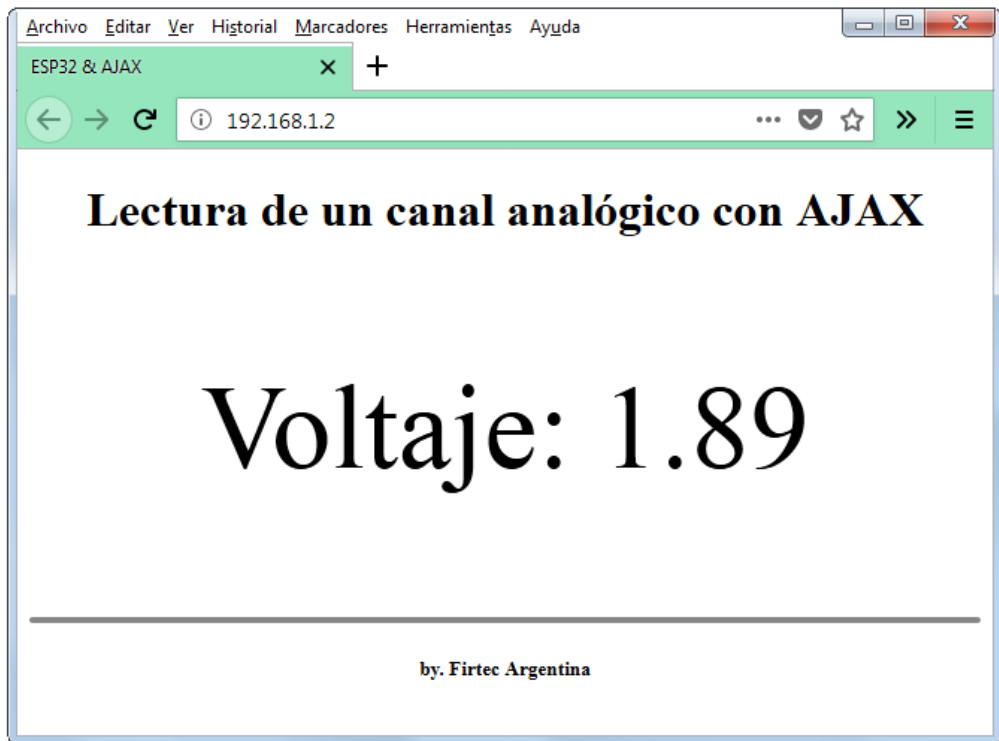
Sabido es que cualquier aplicación decente que pretenda competir en el campo del “Internet de las Cosas” o IoT debe tener conectividad a nivel de red.

En las páginas siguientes veremos una serie de ejemplos que logran justamente eso.

Construiremos Servidores Web simples con HTML, otros bajo el control de Ajax y enlaces por UDP, el llamado “RS-232” de Internet.

Y si es engorroso conectar una pantalla gráfica, porque no construir una “*pantalla*” que se pueda ver desde cualquier lado y con cualquier equipo que tenga un navegador.

Observe la siguiente imagen.



Un clásico voltímetro mostrado en un navegador. No hemos agregado gráfica en este ejemplo pero podríamos construir una gráfica tan bonita como conocimientos de diseño tengamos. (O encargar el trabajo a quien sepa de diseño web).

Básicamente esa es la idea, pensar diferente, convertir eventualmente lo que el usuario tiene, tablet, teléfono, etc en la pantalla gráfica que la electrónica necesita para visualizar datos o controlar eventos.

Alimentación para el ESP32 Thing.

Las dos entradas de alimentación principales para ESP32 Thing son del USB o una batería de litio. Si tanto el USB como la batería están conectados a la placa, el controlador cargará la batería a tomando de la fuente unos 500 mA. El regulador de 3.3V en el ESP32 Thing puede suministrar de manera confiable hasta 600mA, lo que debería ser más que suficiente para la mayoría de los proyectos.

El ESP32 puede extraer hasta 250 mA de la fuente de alimentación durante las transmisiones de RF, pero en general el consumo está en el orden de 150 mA, incluso mientras se transmite activamente a través de WiFi.

La salida del regulador también se divide en los lados de la placa en los pines



Dentro de la cabecera (*HEAD*), lo que se incluye aquí se muestra en la barra del título o la ventana del navegador.

`<BODY> ... </BODY>`

Delimita y engloba el cuerpo de la página, que son el conjunto de informaciones (texto e imágenes) que se muestran en la página, así como las indicaciones de cómo deben mostrarse.

Formatos de párrafo.

El texto de la página se puede estructurar en encabezamientos de los diferentes apartados de la página, que pueden tener distintos niveles de 1 a 6 (siendo 1 el más importante) y párrafos normales.

`<H1> ... </H1>` o `<H2> ... </H2>` (hasta 6)

Párrafos que son encabezamientos (con distintos niveles).

`<P>... </P>`

Párrafos normales.

`<P align="center">... </P>`

Permite alinear el texto del párrafo. Se puede aplicar igual a las etiquetas

`<H1>`, `<H2>`, etc ...

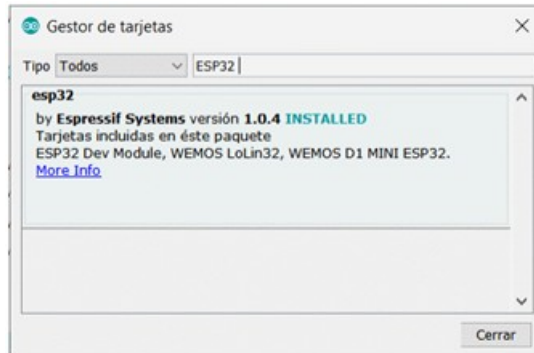
`
`

Permite partir un párrafo empezando una línea nueva pero sin dejar espacio.

`<HR Size=7 noshade/>`

Pone una línea horizontal de separación con un espesor definido y sin sombra.

Formatos de texto.



Instalamos y es recomendable cerrar y reiniciar nuevamente el programa Arduino.

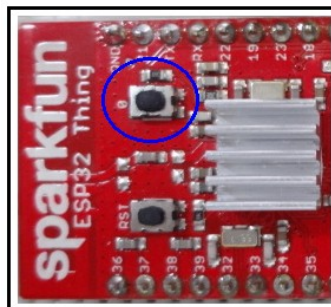
En alguna computadoras *Windows* se da que los nuevos valores de configuración son tomados cuando el programa reinicia, si bien es una situación muy puntual puede ocurrir.

Como programar la placa ESP32 Thing.



El IDE de Arduino tiene dos Iconos o botones, uno sirve para solo compilar el proyecto y ver si existen errores, el otro compila y programa la placa. Para programar la placa actuamos sobre el botón correspondiente y cuando el propio IDE lo indique debemos pulsar unos segundos el botón de programación en la placa ESP32.

Si no mantenemos apretado el botón en la placa salta un error indicando que no se encuentra la placa a programar, tendremos que iniciar todo el proceso de nuevo, compilación y programación.



Botón de programación en la placa ESP32 Thing.