Lenguaje C orientado a Microcontroladores

Rev.1.0
2011

Preparado por: Juan Ignacio Huircán

Dado que el documento es una primera revisión no todas las fuentes salvo:

Microchip, PIC16F87XA Data Sheet 28/40/44-Pin Enhanced Flash Microcontrollers

B Knudsen, C Compiler for the PICmicro Devices Version 3.2 User's Manual

Lenguaje C orientado a Microcontroladores

Review

Un programa codificado en lenguaje C resulta muy útil en la aplicación de µControladores, dado que su compilación es bastante eficiente y óptima acercándose a la codificación de lenguaje de máquina. Lo descriptivo de la sintaxis permite elaborar de mejor forma los algoritmos olvidándose de los molestos *push* y *pop* usados en el lenguaje de máquina cuando se usan saltos a subrutinas. En la primera parte se plantean elementos genéricos de lenguaje C, pero siempre mirando las posibles aplicaciones con µControladores.

Elementos básicos

Comentario, este permite la documentación del código y se usa de acuerdo a la siguiente sintaxis

```
/* Este es un comentario */
o
// Este es un comentario
```

Inicio y *fin* de bloque, permite agrupar un número de instrucciones las que pueden ser ejecutadas con cierta prioridad. Se usa { para iniciar bloque y } para finalizar bloque.

```
{  // inicio de bloque

   // instrucciones

}  // final de bloque
```

Identificador, es el nombre que se le da a una variable o función por lo general asociado al tipo de dato que ha de contener la variable o al tipo de procedimiento que ha de realizar la función.
Tipo, es una palabra reservada definida que indica el tipo de variable que se ha de definir y su alcance numérico, esto de acuerdo a la Tabla 1.

Tabla 1.

Tipo	Código	Tipo de compilador C
char	Entero de 8 bit	Estándar
int	Entero de 16 bit	Estándar
long	Entero de 32 bit	Estándar
float	Real de codificado en 32 bit	Estándar
double	Real codificado en 64 bit	Estándar
uns16	Entero sin signo 16 bit	CC5x (*)
uns32	Entero sin signo 32 bit	CC5x (*)

(*) Ver manual de CC5X compiler.

Así, la definición de las variables se hace usando el *tipo* y un *identificador* que da el nombre de la variable finalizado por un ; (punto y coma). También puede definir varias variables con un solo tipo. En esta caso son separadas por , (coma) pero siempre se finaliza con ; (punto y coma).

```
char i; // define variable tipo char de 8 bits
char j,i;
float x,r;
long a,b,c,d;
int i,j; // define dos enteros
```

Estructura básica de un programa

La estructura de un programa básico en lenguaje c se indica en el esquema de la Fig. 1, el cual muestra un bloque principal llamado `main()` y bloques de funciones. Puede ser, dependiendo de la aplicación, que solo se requiera del bloque principal.

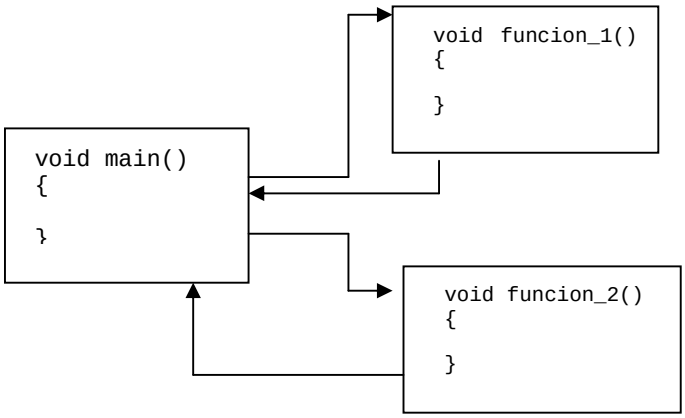


Fig.1. Esquema de un programa

El siguiente código fuente contempla la declaración de las variables y el módulo principal.

```
// Ejemplo de programa sin funciones
// Aquí se definen las variables globales

void main()
{
   // aquí se definen las variables locales
   // aquí van las instrucciones
}
```

Una variación permite incorporar la declaración de las variables y el módulo principal y la zona donde se recomienda se escriban las funciones.

```
// Ejemplo de programa con funciones
// Aquí se definen las variables globales
// Aquí se Escriben las funciones

void main()
{
   // Aquí se definen las variables locales

   // Aquí van las instrucciones y llamados a funciones
}
```

Instrucciones básicas

Estos son los elementos más básicos para la generación de un programa, se considera: asignación, operaciones matemáticas, operaciones lógicas.

Asignaciones

Consiste en asignarle un valor a una determinada variable. En el ejemplo se declaran dos variables de 8 bits y luego se le asignan los valores respectivos.

```
char i, j; // Declaración de variables
void main()
{
   i=10;
   j=20;

  // Aquí van otras instrucciones
}
```

Los valores asignados a las variables tipo `char` o `int` pueden tener distintos formatos ya sea decimal, octal, hexadecimal o binario. Esto es muy útil dado que en el más bajo nivel siempre se trabaja en binario.

Independencia de la representación

A nivel de un µControlador, todas las instrucciones de trabajan en binario, por lo tanto, los datos también, es decir, se puede representar un valor en un formato numérico, ya sea hexadecimal, decimal o tipo carácter, pero a bajo nivel es siempre binario. Por ejemplo, sea la variable dato, la cual se le asigna el valor 49. Este valor será almacenado en memoria en formato binario como 00110001, lo que representado en formato hexadecimal corresponde al 31H.

```
void main()
{
 char dato;
 dato=49;
}
```

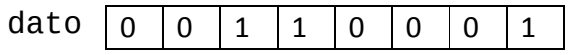


Fig.2. Almacenamiento del dato en memoria.

En el caso de trabajar con 16 bit, el formato hexadecimal quedará 0x0031.

Formatos de asignación

La asignación puede ser hecha en decimal, hexadecimal, binario o carácter (si es que se conoce). La Tabla 2, indica el formato binario para el compilador CC5X.

Tabla 2.

Decimal	Binario(*)	hexadecimal	ASCII
10	0b00001010	0x0a 0x0A	LF
123	0b01111101	0x7D 0x7d	'}'
49	0b00110001	0x31	'1'
65	0b01000001	0x41	'A'

Note que el número codificado hexadecimal comienza con 0x, es decir 31H será 0x31 y el binario comienza con 0b, es decir, 00110001 será 0b00110001 (este segundo formato corresponde al entregado por el

compilador CC5X). El caso de las variables tipo `char` se tiene el siguiente ejemplo, se inicializa una variable con el valor 1 ASCII, sin embargo esto se puede realizar de varias formas.

```
char c;
void main()
{
   c=0x31; // en hexa
   c=49;   // en decimal
   c='1';  // en caracter
}
```

```
int d;
long x;
void main()
{
   d=0xA031; // en hexa
   x=49000L; // en long
}
```

Operaciones Matemáticas

Se tienen la suma, resta multiplicación y división, división entera más las operaciones de incremento y decremento.

```
char i,j, k;
void main()
{
   i=10;
   j=20;

   k=i+j; //suma de 2 enteros
   k=i-j; // resta de 2 enteros
   k=i*j;  // multiplicación de 2 enteros
   k=i%j; // Division entera (debe dar el resto)
   k++;  // incremento k=k+1
   k--;  // decremento k=k-1;
}
```

Operaciones Lógicas

Se realizan con variables enteras y afectan directamente al bit del dato, se tienen los operadores lógicos y los desplazamientos a la Izquierda y a la derecha.

```
char i,j, k;
void main()
{
  k=i&j; // operación AND entre i y j bit a bit
  k=i|j; // operación OR entre i y j bit a bit
  k=~i;  // operación negación de i asignado a k
  k=i<<1; // i desplazado en un bit a izquierda
  k=i<<2; // i desplazado en 2 bit a izquierda
  k=i>>1; // i desplazado en un bit a la derecha
  k=i<<2; // i desplazado en 2 bit a la derecha
}
```

Sentencias de control

Permiten definir las funciones a ejecutar dependiendo de ciertas condiciones que se deben evaluar. Estas condiciones pueden ser tipo *lógicas*, tipo *mayor*, *menor* o *igual* o *ambas*. Por lo general se agrupan entre paréntesis para definir la prioridad de la evaluación de la condición.

Tabla 3.

Operador de condición	Descripción
==	Igual
!=	Distinto
> <	Mayor, menor
>= =<	Mayo o igual, Menor o igual
&&	Operador Y
||	Operador O

La condición verdadera puede ser también de la siguiente forma: Si se quiere preguntar por una variable que es mayor que cero solo se coloca la variable si se quiere preguntar si es falsa entonces se niega, usando el operador !.

Sentencia `if-else`

Puede usarse el `if` solo o el `if-else`. Si la condición es verdadera entonces se ejecuta una instrucción o un grupo de instrucciones en caso contrario se ejecuta la que sigue.

```
char i,j,k,h;
void main()
{
  i=10;
  j=20;
  if(i==j) k=1; // si i=j entonces k=1 sino k=2
  else k=2;

  // s i=10 entonces k=20,j=10 sino k=40,j=20,h=60
  if(i==10) {
              k=20;
              j=10;
            }
  else
       {
              k=40;
              j=20;
              h=k+j;
       }

 // Si i=10 y j=20 entonces...
 if((i==10) && (j==20)) {
                          k=20;
                          j=10;
                         }
 if(i) k=20; // si i es mayor que cero
 if(!i) k=20; // si i es cero
}
```

Sentencia `while`

Esta evalúa la condición al inicio, si ésta es verdadera, entonces se ejecuta el bloque que viene a continuación en caso contrario lo salta. El formato es el siguiente:

```
while(condición) instrucción;
```

```
while(condición) {
                  instrucción_1;
                  ...
                  instrucción_n;
                 }
```

En el siguiente código fuente se muestra que el ciclo `while` se realiza ya que i siempre sea mayor que 0. Sin embargo en algún momento i será menor que 0, dado que k se va decrementando y se hace igual a i dentro del ciclo.

```
char i,j,k,h;
void main()
{
  i=1;  k=40;  j=10;
  while(i>0) // Solo mientras i>0 hace lo que viene a cont.
  {
       h=k+j;
       k--;
       i=k;
  }
}
```

En el ejemplo sólo se evalúa la condición verdadera, pero nunca sele del ciclo.

```
char i,j,k,h;
```

```
void main()
{
  i=1;
 // Solo mientras i mayor que cero hace lo que viene a cont.
  while(i)   // Puede usarse while(1)
   {
       k=40;
       j=20;
       h=k+j;
   }
}
```

Para salir de un ciclo donde no existe posibilidad de que la condición sea alterada, se puede usar la sentencia `break.`

```
char i,j,k,h;
void main()
{
  i=0;
  while(1)
  {
       i++;
       if(i==5)break; // Sale cuando i=5
   }
}
```

La sentencia `do-while` se evalúa la condición al final, por ejemplo “Hágalo mientras se cumpla la condición”.

```
void main()
{
 char i;
 i=10;
 do
 {
    i--;
   }while(i!=5); // Sale cuando i=5
 }
```

Sentencia `switch - case`

Esta instrucción se usa cuando se deben seleccionar entre varias opciones tipo numéricas. Es básicamente un selector.

```
char c;
void main()
{
 // se le debe asignar el valor a la var. c
  switch(c)
  {
   case 1: // debe saltar a función
            break;
   case 2: // debe saltar a función
            break;
   case 3: // debe saltar a función
            break;
  }
}
```

Sentencia `for`

Permite el uso de los clásicos ciclos `for`. El formato es el siguiente:

```
for(expr_1;expr_2;expr_3)instrucción;
```

```
for(expr_1;expr_2;expr_3)
{
  instrucción_1;
  ...
  instrucción_n;
}
```

Tabla 4.

	Descripción
expr_1	Condición de comienzo, pueden haber varias inicializaciones separadas por coma.
expr_2	Se efectuará el bucle mientras se cumpla esta condición, se evalúa la condición antes de entrar al bucle.
expr_3	Indica lo que hay que hacer cada vez que se termina el bucle, puede haber varias instrucciones separadas por una coma.

El código fuente muestra que la variable i se inicializa en cero, una vez realizadas las instrucciones dentro del bloque, en este caso, hacer `j` igual a cero, se ejecuta la instrucción incremento en uno para la variable `i`.

```
char i,j;
void main()
{
   for(i=0;i<10;i++) // inc. i desde 0 mientras sea menor que 10
   {
     j=10; // lo repite tantas veces de acuerdo a la condicion
   }
}
```

Funciones y Llamados a funciones

Las funciones permiten agrupar código y optimizar el tamaño de éste. Se tienen distintos tipos de funciones:

- ☐ Las que solo ejecutan tareas y no devuelven valor.
- ☐ Las que ejecutan y devuelven un valor.
- ☐ Las que reciben parámetros y no devuelven.
- ☐ Las que reciben parámetros y devuelven información.

El traspaso de información a una función se realiza a través de sus argumentos. Por lo general se conoce como *paso de parámetros por valor*, esto quiere decir que se puede pasar el número o dato en forma explícita o a través de una variable. Cuando el valor se pasa a través de alguna variable, si el contenido de la ésta es modificado en el interior de la función, el valor de la variable fuera de la función no es modificado.
El *paso de parámetros por referencia* solo se puede realizar usando punteros, en este caso, si el dato es pasado mediante una variable a la función, y éste es modificado al interior de ella, su modificación se verá reflejada fuera de la función. El uso de punteros en este caso como argumento permite manejar dicha situación, ya que lo que se está transfiriendo a la función es la dirección donde se encuentra el dato.

Funciones que no usan argumentos
Estas funciones son identificadas con un nombre de acuerdo a la tarea que realizan, el identificador utilizado obedece al mismo esquema en el cual se declaran las variables. Sin embargo, el identificador de las funciones lleva paréntesis `()`. Habitualmente son declaradas antes del `main()` antes de ser utilizadas, pero para ahorrarse este proceso por lo general se escriben sobre el `void main()`.
Las funciones que no devuelven valor son declaradas usando la palabra `void` junto al nombre de ésta, los paréntesis y el punto y coma ;. Son llamadas directamente a través del nombre o identificador correspondiente. La siguiente función primero se declara para luego ser llamada desde el programa principal no recibe argumento ni tampoco retorna un valor.

```
void activar_sensores(); // Declaración

void main()
{
  activar_sensores(); // salta a la función correspondiente
}

void activar_sensores()
{
  // instrucciones
}
```

En el caso de no recurrir a la declaración, es escribe la función sobre el `void main()`.

```
void activar_sensores()
{
  // instrucciones
}

void main()
{
  activar_sensores(); // salta a la funcion correspondiente
}
```

Las que devuelven valor deben ser llamadas a través de una asignación, en donde el nombre de la función es asignado a una variable que recibirá el resultado devuelto por la función. La función en este caso debe ser declarada (o definida) considerando el tipo de valor que ha de retornar o devolver. La siguiente función es declarada tipo `char`, retorna el valor correspondiente, por lo tanto al ser llamada, la función inmediatamente debe ser asignada a una variable del mismo tipo a ser retornada.

```
char lee_sensores();
```

```
void main()
{
  char sensor;
  sensor=lee_sensores(); // salta a la funcion y retorna el valor en la variable sensor
}

char lee_sensores()
{
  char x;
  // Instrucciones
  // Se debe asignar el valor de x en alguna parte
  return(x); // devuelve el valor de x tipo char
}
```

Se repite el procedimiento, pero esta vez se omite la declaración, pero se escribe la variable antes del `main()`.

```
char lee_sensores()
{
  char x;
  // Instrucciones
  // Se debe asignar el valor de x en alguna parte
  return(x); // devuelve el valor de x tipo char
}

void main()
{
  char sensor;
  sensor=lee_sensores(); // salta a la funcion y retorna el valor en la variable sensor
}
```

Para el paso de los argumentos, estos son definidos entre los paréntesis de la función considerando el tipo de la variable requerida. La siguiente función recibe un parámetro tipo `char` por valor y no retorna valor.

```
void activa_motor(char num_motor)
{
    // instrucciones
}

void main()
{
  char sensor;
  activa_motor(1); // pasa el valor 1 haciendo ref al num de motor
}
```

El siguiente ejemplo muestra una función que recibe parámetros y retorna un valor. El valor retornado a su vez puede ser usado por una sentencia de control para tomar alguna decisión.

```
char enviar(char dato)
{
  char status;
 // codigo
 return (status);
}

void  main()
{
 char estado_com;
 while(1)
 {
    estado_com=enviar(0xaa);
    if(estado_com!=1) break;
 }
}
```

Punteros

Los punteros son variables que almacenan la dirección de una variable, el contenido de dicha variable es accesado en forma indirecta. La variable tipo puntero se define en forma similar a una variable que almacena datos, pero se agrega un *.

```
char *p; //p es un puntero a un char
char i, *p; // i es un char y p es un puntero a char
int j, *p; // j es una var entera y p es un puntero a entero
```

¿Qué quiere decir puntero a `char` o puntero a `int`?. Simple, cuando un puntero a `char` se incrementa, la dirección almacenada en la variable se incrementa en una posición de memoria en cambio en un puntero a `int` se incrementa de 2 posiciones. En el caso de que fuera a una variable tipo `long`, la dirección se incrementaría en 4 posiciones de memoria.

Para saber la dirección de memoria de una variable y asignarla a un puntero se usa el operador & y el nombre de la variable o solo el nombre de la variable.

```
char i, *p; // i es un char y p es un puntero a char
void main()
{
  p=&i; // p almacena la dirección de memoria donde esta i
}
```

Para saber el contenido de la variable apuntada por el puntero se usa el operado *.

```
char i, j, *p; // i y j char y p es un puntero a char
void main()
{
  i=10; // i contiene el valor 10
  p=&i; // p almacena la dirección de memoria donde esta i

  j=*p; //El contenido de p es almacenado en j
}
```

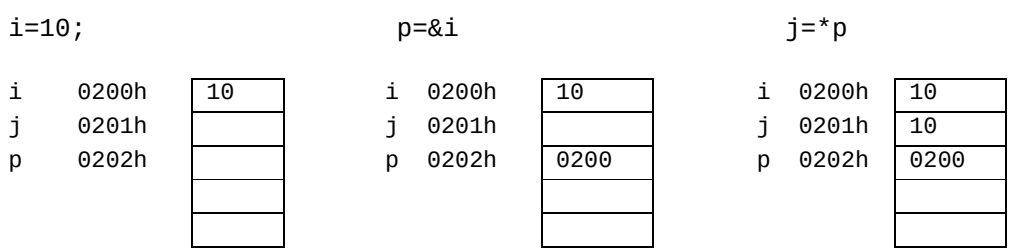


Fig.3. Manejo de puntero.

Uno de los usos de los punteros es para el paso de variables por referencia a una función, es decir, el valor de la variable que se pasa como argumento a la función cambia si es que en el interior de la función cambia. Esto ocurre a que en vez de pasar el valor del argumento a la función, se pasa su dirección. Esto se podrá realizar si se utiliza un puntero. El siguiente ejemplo es solo ilustrativo, ya que si se va a devolver un valor, puede usar la instrucción `return()`.

```
void duplicar(char *p)
{
   *p=(*p)*2;
}
```

```
void main()
{
 char dato, *pdato;
 dato=10;
 pdato=&dato;
 duplicar(pdato);
}
```

Arreglos y matrices

Los arreglos son vectores unidimensionales (llamados *string* o cadenas) o bidimensionales (Matrices). Su formato es el siguiente:

```
tipo ident_arreglo[num_elementos];
tipo ident_arreglo[num_fila][num_col];
```

A continuación se muestran ejemplos de arreglos unidimensionales y bi-dimensionales

```
char cadena[10];// define una var string llamada cadena de 10 char
int  datos[15]; // define una var string llamada datos de 15 int
char cad[2][3]; // define una matriz cad de 2 filas y 3 col
```

Los arreglos pueden contener números o caracteres, pueden ser usados para manejo de cadenas de caracteres o string. Es conveniente siempre inicializar con un valor los arreglos. El primer elemento del arreglo corresponderá al almacenado en la posición 0 y el *n*-ésimo elemento corresponderá al almacenado en la posición n-1 del arreglo. Cuando se definen los arreglos, estos ocupan zonas completas de memoria, con direcciones consecutivas.

Inicialización de arreglos

Los arreglos pueden inicializarse elemento a elemento si es que es pequeño, o a través de la inicialización usando la cadena completa.

```
char data[5],i;
void main()
{
  for(i=0;i<5;i++) data[i]=0; // rellena con 0s
}
```

Los arreglos pueden ser inicializados con mensajes, estos son formados por cadenas de caracteres llamados *cadenas* o *string*. Habitualmente, se usan las comillas para establecer o limitar el mensaje.

```
// Esta inicialización corresponde al Turbo C
char msg[10]={"HOLA"};

// C estándar
char x[7]={"Juan\xd"};
char c[]={"popo"};
```

En los compiladores C estándar, cada cadena termina con el carácter 0x00 a continuación del último carácter. De esta forma en el ejemplo de la cadena `msg`, el elemento 5 de la cadena debe llevar un carácter 0x00.

H	O	L	A	0x00				
msg[0]	msg[1]	msg[2]	msg[3]	msg[4]				

0x48	0x4F	0x4C	0x41	0x00	¿?	¿?	¿?	¿?

Las funciones más usadas en el manejo de cadenas son la copia de cadenas o la concatenación de cadenas.

Las cadenas comúnmente pueden ser usadas con caracteres de control tales como el 0x0d (CR) o el 0x0a (LF). Estos caracteres pueden ser incorporados al final de la cadena en forma directa en la posición correspondiente o de la siguiente forma:

```
char msg[10]={ "HOLA\xd\xa"};
```

0x48	0x4F	0x4C	0x41	0x0d	0x0a	0x00	0x00	¿?

Arreglos y punteros

Los punteros pueden ser inicializados con la dirección inicial del arreglo, de esta forma se tiene

```
char cad[10], *pcad;

//Formas equivalentes

pcad=&cad[0];
pcad=&cad;
pcad=cad;
```

Los punteros pueden usarse para apuntar al primer elemento de una cadena y se pueden inicializar de acuerdo al siguiente ejemplo.

```
char *p="chao";
```

Pasando cadenas a funciones

Habitualmente se utilizan punteros, debido a que se pasa la dirección del primer elemento de la cadena, evidentemente, el puntero debe ser del mismo tipo del dato apuntado, de tal forma que cuando se incremente el puntero, se incremente en una posición dentro del arreglo. En C estándar se tiene que se asigna un puntero al primer elemento de la cadena, esto permite pasar filas completas.

El siguiente ejemplo permite traspasar un mensaje ASCII a una cadena. Este ejemplo compila en C estándar y CC5X.

```
void copy_msg(char *pdest, const char *porig)
{
  char x;
  while(1)
  {
      x=*porig;
      *pdest=x;
      if(*porig==0x00) break;
      pdest++;
      porig++;
  }
}

void main()
{
 char msg[8];
 copy_msg(msg,”HOLA”); // Copia en la variable msg la cadena HOLA
}
```

Copiando del contenido de una cadena en otra, está codificado para CC5X

```
void copy_str(char *pd, char *po)
{
  char x;
  while(1)
  {
      x=*po;
      *pd=x;
      if(*po==0x00) break;
      pd++;
      po++;
  }
}
```

```
void main()
{
 char  msg[8],cad[8];

 cad[0]='C';
 cad[1]='H';
 cad[2]='A';
 cad[3]='O';
 copy_str(msg,cad);
}
```

Pudiera el quinto caracter no ser 0x00, para evitar esto, se agrega en la inicialización.

```
char msg[10]={“HOLA\x0”}; // Este es para C estándar
char *pcad=”HOLA\x0”;
```

El ejemplo siguiente permite traspasar una cadena a una función.

```
void enviar_msg(char *p)
{
 char i;
 i=0;
 while(i)
  {

  }
}

void main()
{
 char msg[10]={“HOLA”};
 char *pm;
 pm=&msg[0];
 enviar_msg(pm);
}
```

El µControlador y el compilador CC5x

Arquitectura del µC

Para las aplicaciones se utilizará un µC basado en la familia 16F87Xa de microchip. Este µC contiene registros, módulos ADC, comunicación serial, temporizadores y manejo de interrupciones. Además posee un memoria de programa de entre 4 y 8Kbytes, su reloj puede ser de entre 4 y 20MHz. (ver data µC16F87xA). Un esquema simplificado es el de la Fig. 4.

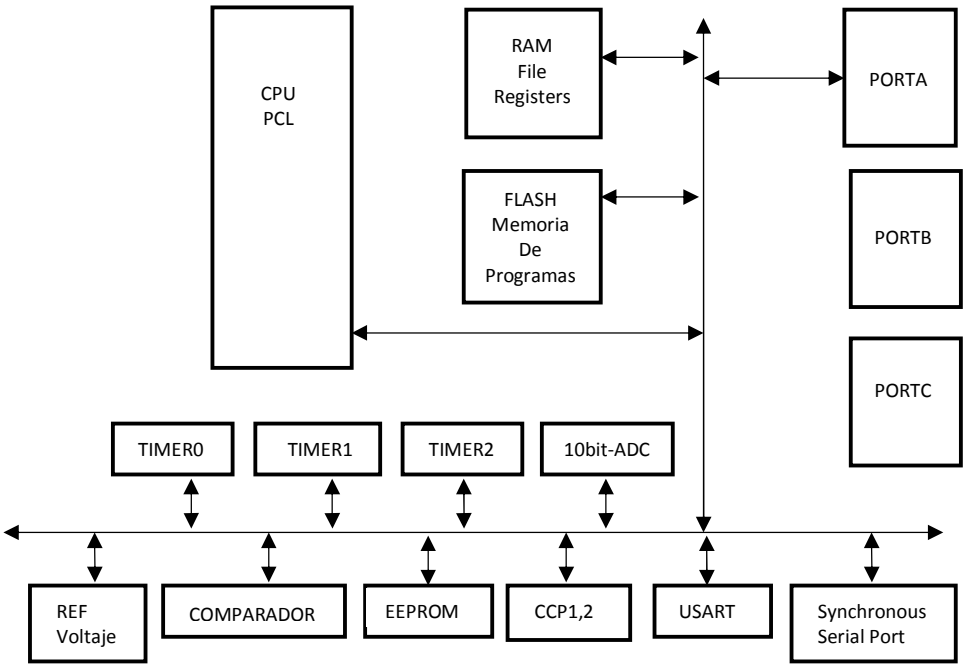


Fig. 4. Esquema simple del µC 16F87XA.

Es bastante sencillo de hacer funcionar, solo basta un par de capacitores, el cristal más un resistor, de acuerdo a la Fig. 5. En dicho circuito, el terminal 28 correspondiente al bit más significativo del puerto B, se usará como monitor del funcionamiento. La tarea básica a implementar será poner en 1 y en 0 alternadamente el bit mencionado de tal forma de visualizarlo mediante un instrumento.

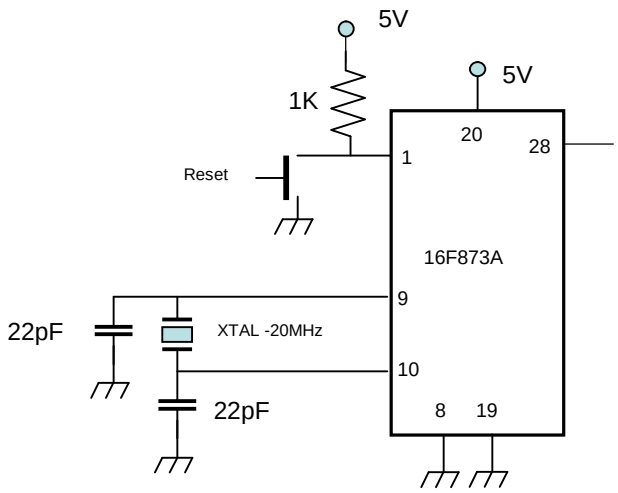


Fig.5. Esquema de para funcionamiento básico.

Entorno de desarrollo

El entorno de desarrollo MPLAB permite el trabajo con el µC incorporando al editor y herramientas de simulación y un compilador proporcionado por otro fabricante, en este caso el compilador C CC5x desarrollado por B. Knudsen (ver www.bknudsen.com). Al trabajar con el compilador CC5x, se debe incorporar el archivo cabecera `16F873a.h`, esto automáticamente permite configurar el entorno de trabajo, quedando definido todos los "recursos" de la CPU, registros y módulos con los nombres identificados en la referencia técnica del µC (ver tutorial). Se sugiere revisar el archivo de cabecera y contrastarlo con los nombres de los registros y direcciones asociadas descritas en el manual.

Los registros de µC vienen definidos en el archivo de cabecera, estos pueden ser accesados directamente al igual que también sus bits individuales. En algunos casos los registros ya tienen definidos algunos bits internos usando otros identificadores, esto no es incompatible entre sí.

Como ejemplo se tiene el caso del puerto B, llamado **PORTB** por el compilador CC5X, el cual está definido en el archivo de cabecera como una variable entera de 8 bits.

También es posible encontrar el nombre de algunos bits definidos como variables tipo bit, ya sean banderas (Flags) o bits de habilitación de periféricos. Un ejemplo es RCIF (**R**e**C**eive **I**nterrupt **F**lag) que está definido como una variable tipo bit o el PEIE (**PE**riferical **I**nterrupt **E**nable).

Definiciones de bit y uso de registros

En el caso del PORTB, se accesan los bit recurriendo a una definición de campo de bits. De esta forma PORTB.0 corresponde al bit menos significativo del puerto y el PORTB.7 corresponderá al bit más significativo. Así se tiene que el PORTB.x, donde x representa la posición del bit dentro de la variable entera. Esto puede ser extendido a las variables enteras de 16 y de 32 bits.

```
void main()
{
  char dato;
  uns16 x;

  dato.2=1; // se setea el segundo bit menos significativo de la variable
  x.15=0;   // reset al bit menos significativo de la var. x
}
```

Las variables tipo bits son uno de los elementos más importante dentro de las aplicaciones montadas sobre µC, en la sintaxis del compilador se utiliza una directiva llamada #pragma.

#pragma bit Indentificador_del_ bit @ elemento_de_bit_en_el_µC

Esto permite asignar una variable de bit a un determinado bit de algún registro existente en el hardware, por ejemplo, sea el bit menos significativo del puerto B, PORTB.0 y este bit se usa para leer el estado de un dato en particular,

```
#pragma bit status_dato @ PORTB.0
```

La variable de bit `status_dato` está asignada al bit `PORTB.0`.

Estas variables pueden ser encuestadas para saber si están en 1 o en 0. La encuesta comúnmente se realiza como operación de decisión en una sentencia de control. Esta operación es válida tanto para una variable definida tipo bit o para un determinado bit de una variable entera.

```
if(PORTB.0==1)
 {
 }
else
{
}
```

```
while(PORTB.2)
{
}
```

```
#pragma bit status_bit @ PORTB.7

void main()
{
  while(status_bit)
   {
   }
}
```

```
if(PORTB.0==0) {
               }
else
{
}
```

```
while(!PORTB.2) // Pregunta si es 0
{
}
```

Consideraciones Especiales

Respecto de las variables enteras

Se debe considerar que las variables enteras se pueden acceder al bit o al nibble en el caso de ser de 8 bit. Las de 16 bit se pueden acceder tanto al bit, de 4, de 8 bit o completas. Esto implica tanto su lectura como su asignación de algún valor.

```
char i;
uns16 dato; //Entero sin signo de 16 bits
void main()
{
  i=10; // 0x00001010 0x0a

  dato=2048; //  0b0000100000000000=0x0800

 //modificando un bit

  i.0=1; // se modifica el bit de menos peso luego i=0b00001011=11

  dato.low8=0xff; //solo se acceden los 8 bit de menos peso asi dato=0x8ff;
}
```

Respecto de las cadenas de caracteres

El compilador CC5X inicializa las cadenas o string de acuerdo al siguiente formato:

```
const char *ps = "Hello world!";

const char a2[] = "ab";
```

Cuando se han de pasar cadenas o *string* a funciones puede usarse la siguiente sintaxis:

```
void funcion(const char *str)
{
  // codigo
}

void main()
{
  char tab[20]; // string en RAM
  const char ctab[] = “A string”;

  funcion(“Hello”); // Pasando el string explicito
  funcion(&tab[i]); // Usando una direccion de la cadena
  funcion(ctab);    // Pasando la direccion del primer elemento
}
```

Manejo de Puertos de E/S

El µC posee al menos 3 puertos que pueden ser usados como entradas-salidas digitales o como entradas análogas (si corresponde). Los Puertos son básicamente registros de 8 bits basados en Flip-Flops Tipo D, que pueden ser usados como entrada o salida. Los nombres de los puertos corresponden a los especificados en la data del µC, así se tiene el PORTB de 8 bits, el PORTC de 8 bit y el PORTA de 5 bits. Dichos bits se configuran como entrada o salida a través de un registro llamado TRIS. Para el caso del Puerto A será TRISA, para el puerto B, se usa el TRISB, etc. Los puertos señalados poseen más de una función. Estos puertos del µC se usan como si fueran variables enteras de 8 bit y pueden accesarse nivel de bit o en forma de bytes.

Primero se debe especificar si el bit es de entrada o salida, esto se hace definiendo el bit correspondiente como salida, esto se hace con el registro TRISx (x es el puerto A, B o C) mediante el set o reset de dicho registro. Cada bit del registro TRISx maneja la entrada o salida del bit correspondiente del PORTx.

```
TRISB.0=0; // Bit 0 del puerto B se define como salida
TRISC.7=1; // Bit 7 del puerto C se define como entrada
```

Puede definirse todo el puerto como entrada o salida o mezcla de ambas situaciones usando un byte completo sobre el registro TRIS correspondiente.

```
TRISB=0x00; // los 8 bit del PORTB como salida
TRISC=0xFF; // los 8 bits del PORTC como entrada
TRISB=0x0f; // Los 4 bit más significativos de entrada,
            // los 4 bit menos significativos como salida
```

La Fig. 6 muestra un diagrama funcional del terminal PORTB.0, el cual es controlado por el bit 0 del TRISB (TRISB.0).

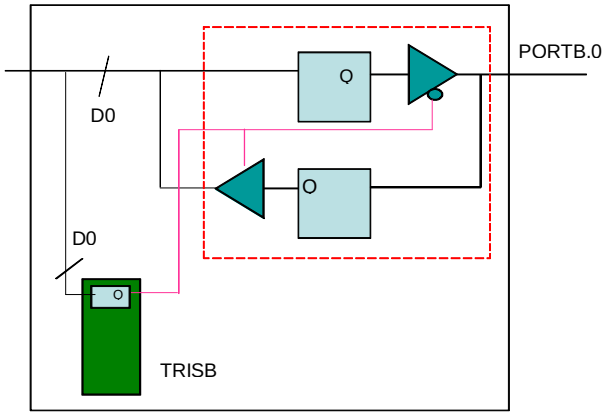


Fig.6. Esquema funcional del PORTB.0.

Programa básico de Test

El programa permite verificar el funcionamiento del µC a través de la generación de un pulso en un terminal del dispositivo. Considerando el bit 7 de puerto B, se tiene.

```
void main()
{
 TRISB.7=0;
 PORTB.7=1;
 while (1)
 {
  if(PORTB.7==1) PORTB.7=0;
  else PORTB.7=1;
 }
}
```

El terminal deberá ser inspeccionado a través de una sonda lógica. Si no se dispone de ella, entonces puede usarse un retardo, el cual en conjunto con un LED permitirá visualizar los cambios de estado.

```
void ret(uns16 r)
{
   while(r>0) { r--;}
}

void main()
{
 TRISB.7=0;
 PORTB.7=1;

 while (1)
 {
  if(PORTB.7==1) PORTB.7=0;
  else PORTB.7=1;
  ret(5000);
 }
}
```

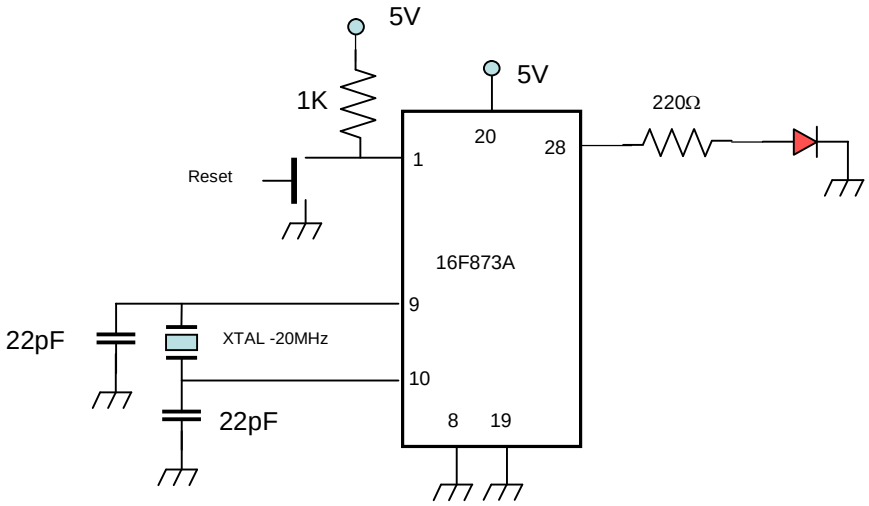


Fig. 7. Circuito de prueba PORTB.7.

El siguiente código permite enviar un byte completo hacia el exterior del µC, primero envía un 0xAA (en binario 0b10101010) que prende bit por medio después de un retardo envía un 0x55 (0b01010101), negando los bit en la salida. Para poder visualizar se recomienda la configuración de la Fig.8.

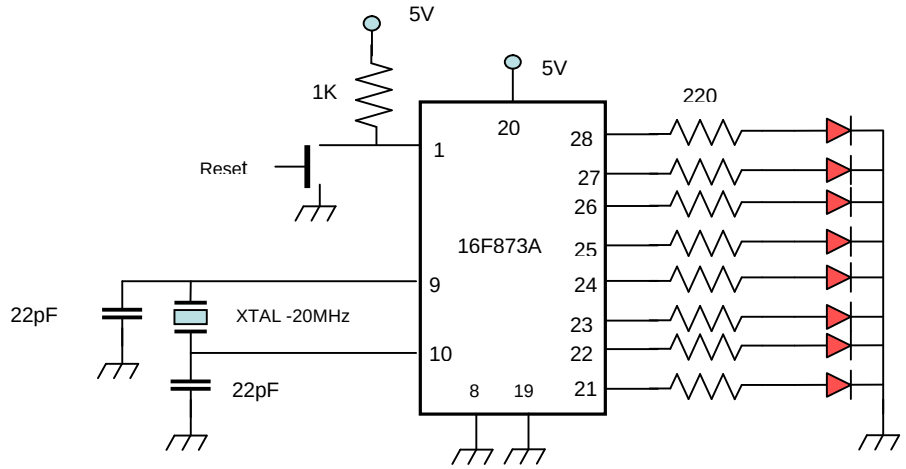


Fig. 8. Circuito de prueba PORTB.

```
void ret(uns16 r)
{
   while(r>0) { r--;}
}

void main()
{
 TRISB=0x00;
 PORTB=0xAA;

 while (1)
 {
  if(PORTB==0xAA) PORTB=0x55;
  else PORTB=0xAA;
  ret(5000);
 }
}
```

Este código puede ser simplificado usando operaciones sobre la variable entera, es decir

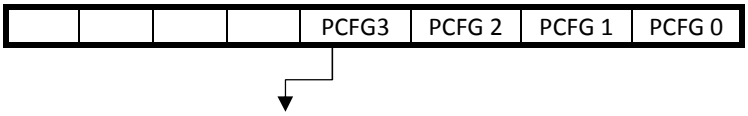
```
void ret(uns16 r)
{
   while(r>0) { r--;}
}

void main()
{
 TRISB=0x00;
 PORTB=0xAA;

 while (1)
 {
  PORTB= ~PORTB; // se niega el contenido del PORTB
  ret(5000);
 }
}
```

Programando el Puerto A

El PORTA requiere de una instrucción adicional, dado que sus terminales posee más de una función su uso como entrada salida digital no es tan directo. Se requiere el uso del registro ADCON1, el que permite definir si los terminales del puerto serán Analógicos o digitales.



	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	VREF+	A	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	D	D	VDD	VSS	3/0
0101	D	D	D	D	D	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	-	-	0/0

Si se han de utilizar todos los terminales del PORTA como I/O digitales, se recomienda usar la configuración PCFG 0111 o 0110.
El PORTA del 16F873A posee solo 5 bits, por lo tanto de acuerdo a las configuraciones de la tabla indicada, no estarían todas disponibles. El siguiente código establece todos los bits del PORTA como salida, adicionalmente va cambiando el bit PORTA.0 de 1 a 0 y viceversa.

```
void main()
{
 ADCON1=0b000001111; // Todos los bit del PORTA como IO digital
 TRISA=0b00011111;   // Todos de salida
 PORTA.0=1;          // Salida puesta en 1
 while(1)
 {
  if(PORTA.0==0) PORTA.0=1;
  else PORTA.0=0;
 }
}
```

Programación Conversor Análogo-Digital

La conversión Análogo- Digital

El proceso de conversión análogo digital consiste en transformar una señal de voltaje analógico en un código o número digital, también llamado cuenta, este proceso tiene varias etapas. De acuerdo a la Fig.9, inicialmente se debe contar con la señal eléctrica que se ha de convertir, posteriormente viene el proceso de muestreo, que transforma la señal continua s(t) en una señal discreta s(k), después el mecanismo llamado ADC (Analog to Digital Converter) realiza la transformación del voltaje analógico asignándole al voltaje de entrada un código binario. Este código puede ser de 8bit, 10bit, 12bit o 16bit. La salida del número binario se puede visualizar en los terminales de salida digital del ADC, de acuerdo al tipo de conversor, éste poseerá por ejemplo 8 terminales de salida si es de 8bits.

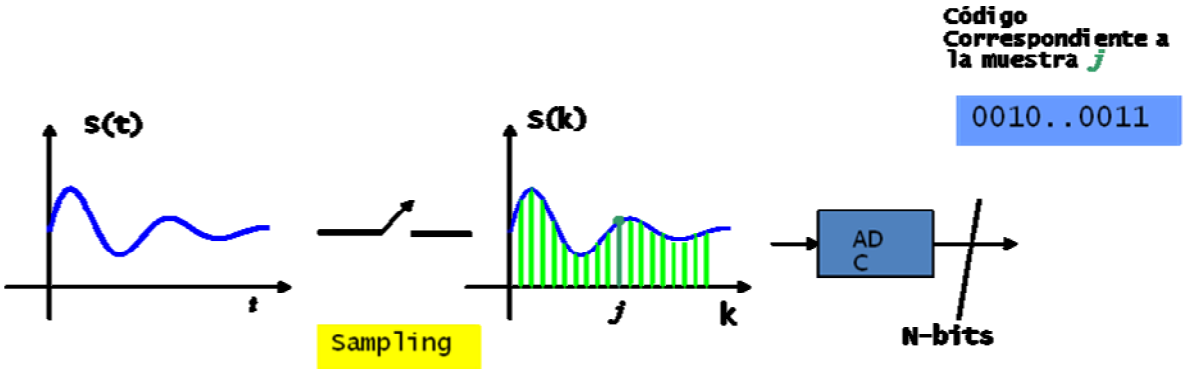


Fig. 9. Proceso de Conversión Análogo- Digital

Internamente el ADC posee distintos mecanismos de conversión, los más comunes son los llamados de aproximación sucesiva, que pueden tener salida paralela o salida serial. El proceso de conversión requiere un tiempo, el que está dado por el tc (tiempo de conversión) del sistema en conjunto con otros elementos que aportan retardos.
De esta forma el tiempo de muestreo ts (Sample Time) que establece cada cuanto tiempo es posible efectuar la digitalización, dependerá del la frecuencia de la señal a digitalizar, el tiempo que se demora el mecanismo en realizar la conversión.

Módulo Análogo-Digital del uC

El módulo de conversión del µC16F87X es de 10 bits, lo que permite una cuenta entre 0 y 1023 (0x000 a 0x3FF). El tiempo de conversión del orden de los 10µSegundos.
El µC16F8XX, posee 5 (8) canales analógicos, los cuales se configuran usando los registros PORTA, ADCON1 y ADCON0. Adicionalmente, pueden usarse los registros asociados a las interrupciones para obtener una temporización más precisa o simplemente leer el dato convertido si tener que encuestar el bit indicador de la conversión.

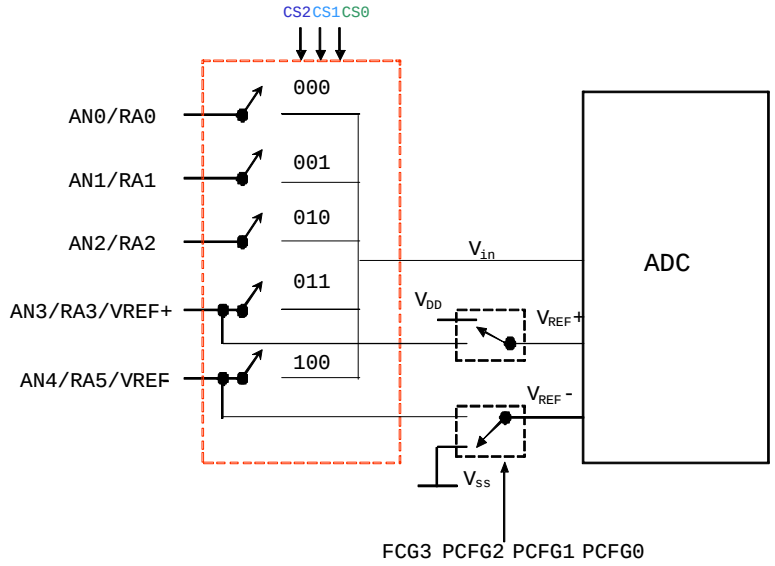


Fig. 10. Módulo ADC.

Configuración

Los canales analógicos se encuentran en el PORTA, mediante el TRISA se definen dichos bits como entrada. Mediante el ADCON0 y ADCON1, los cuales definen cuales serán las entradas analógicas del PORTA en conjunto con la velocidad y el formato de la data (justificación a la derecha o la izquierda). La data resultante de la conversión se almacena en dos registros de 8 bits, ADRESH y ADRESL, de los cuales solo se usan 10 bits.

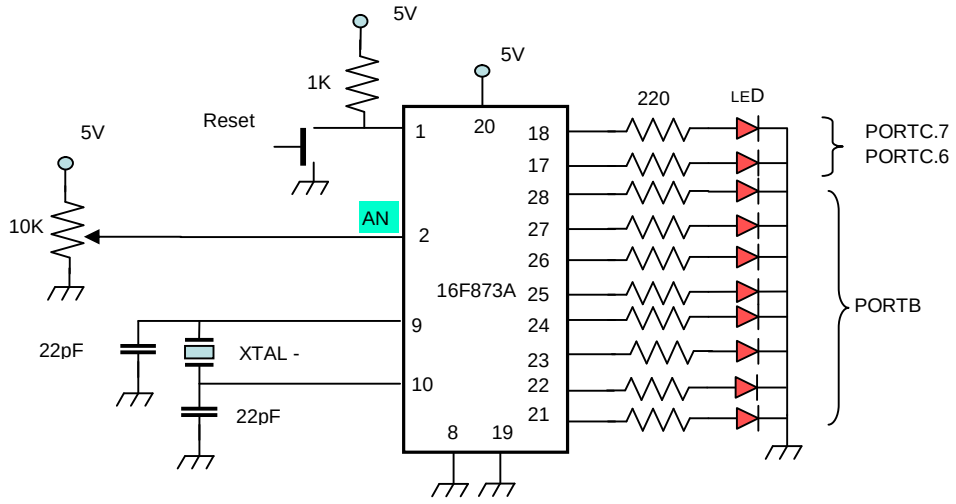


Fig. 11. Plataforma para probar el ADC.

Mediante un simple potenciómetro se puede verificar el ingreso de una señal analógica la cual se verá en los terminales de salida a través de su equivalente digital. La programación está dada de acuerdo a la configuración de los siguientes registros:

- Se define que canales analógicos a utilizar, los que están asignados al PORTA, para ello se configura el registro ADCON0.
- Se programa la frecuencia de operación del modulo, se usan los bits
- Se especifica la justificación de la data
- Se habilita el conversor ADON=1
- Se da inicio a la conversión GO=1
- Se espera que la conversión ha terminado. Se encuesta el bit GO, el cual pasa a 0 cuando la conversión está lista.
- Se leen los datos de los registros ADRESH:ADRESL y se almacenan en la variable que se ha definido para ello.

La siguiente función permite testear el canal cero del conversor, la cual es llamada desde el programa principal. La conversión es realizada por la función `test_adc()`, la cual devuelve el valor convertido en la variable dato, dicha variable es visualizada en los terminales del PORTB y parte del PORTC. Esto dado que se requieren 10 bits. Este proceso de repite continuamente.

```
//Programación Conversor Análogo Digital

           uns16 test_adc()
           {
              uns16 d;
              ADON=1;
              GO=1;  // inicia conversión
              while(GO==1) {
                           } // espera el DONE
              d.high8=ADRESH;
              d.low8=ADRESL;
              return(d);
            }

           void main()
           {
            uns16 dato;

            TRISB=0x00;
            TRISC.7=0;
            TRISC.6=0;
            TRISA=0xff;
            ADCON0=0b10000000; // Configurando el ADC
            ADCON1=0b10000000;

            while(1)
            {
              dato=test_adc();
              PORTB=dato.low8;
              PORTC.7=dato.9;
              PORTC.6=dato.8;
            }
           }
```

El programa test.c verifica el funcionamiento del módulo ADC. Se debe considerar que no existe una temporización respecto del cada cuanto tiempo se produce el proceso de conversión.

Manejo de interrupciones

El manejo de las interrupciones en un µC puede resultar complejo, sin embargo, su implementación se ve simplificada dado que el fabricante del compilador usado provee un esquema basado en una función RSI (Rutina de Servicio de Interrupción) genérica la cual puede ser fácilmente modificada de acuerdo a los requerimientos del usuario.

Cuando alguna de la fuentes de interrupción se activa, el programa en ejecución salta a la dirección `0x0004` (vector de interrupción), en esa dirección debe encontrarse la RSI. Dado que como pueden ser distintas las fuentes de interrupción y solo una función para atenderla, se debe recurrir al uso de los **I**nterrupt **F**lag (banderas de interrupción), los cuales están asociados a las distintas fuentes de interrupción.

Cada interrupción para su adecuado funcionamiento, requiere una habilitación y un proceso de configuración en algunos casos. Para ello se usan los registros OPTION_REG (en CC5x es el OPTION), INTCON, PIE1 y PIR1. Adicionalmente PIE2 y PIR2 para otros periféricos.

Flag	Registro	Declaración bit (CC5x)	Fuente de Interrupción
TMR0IF	INTCON.2	`T0IF`	Timer 0
INTF	INTCON.1	`INTF`	Terminal Int
RBIF	INTCON.0	`RBIF`	4 bit más significativos del PORTB
RCIF	PIR1.5	`TXIF`	Transmisión de datos del UART
TXIF	PIR1.4	`RCIF`	Recepción de datos del UART
ADIF	PIR1.6	`ADIF`	Conversión del ADC

Los bits de habilitación terminan con las letras **I**nterrupción **E**nable.

Bit	Registro	Bit Declaración (CC5x)	
GIE	INTCON.7	`GIE`	Global Interrupt Enable
PEIE	INTCON.6	`PEIE`	Periferical Interrupt Enable
TMR0IE	INTCON.5	`T0IE`	Timer 0 Interrupt Enable
INTE	INTCON.4	`INTE`	External Interrupt Enable
RBIE	INTCON.3	`RBIE`	RB Port Change Interrupt Enable
RCIE	PIE1.5	`RCIE`	Receive Interrupt Enable bit
ADIE	PIR1.6	`ADIE`	Analog to Digital Interrupt Enable

El registro OPTION_REG permite configurar algunas opciones de las Fuentes de interrupción, de esta forma se tiene la siguiente tabla.

Bit definido por el fabricante	Registro OPTION (CC5x)	Declaración (CC5x)

RBPU	OPTION.7	Habilita pull-up PORTB
INTEDG	OPTION.6	Canto de la Interrupción por RB0/INT
T0CS	OPTION.5	Selecciona la fuente de clock de TMR0
T0SE	OPTION.4	Selecciona canto de la fuente del TMR0
PSA	OPTION.3	Asignación de Pre-escalar
PS2:PS0	OPTION.2 OPTION.1 OPTION.0	Codificación del pre-escalar con 3 bits desde 000-1:2 hasta 111-1:128

Se debe incluir una biblioteca especial `int16XXX.h`, la cual permite el manejo de las interrupciones. Dado que existen 3 fuentes de interrupciones se deben configurar los bits de dichas fuentes como entrada.
La programación se basa en lo siguiente, primero se configura en el módulo principal la fuente de interrupción a utilizar. Una vez finalizado este proceso se habilitan las interrupciones a través de los flags correspondientes. El µC procederá entonces a realizar las tareas habituales que no requieren de interrupción, habitualmente se usa un loop infinito mediante un ciclo `while(1)`. Cuando ocurre la interrupción, se setea la bandera correspondiente a la fuente de origen y el programa salta a la dirección `0x0004`. En dicha dirección se encuentra la RSI, dentro de dicha función se debe inicialmente salvar los registros actuales, debe encuestarse el flag correspondiente a la fuente de interrupción, dado que está en 1, mediante un `if` es sencillo, sin embargo, después de terminada la atención debe hacer un reset al flag, para permitir una nueva interrupción. Al terminar de ejecutar la RSI, esta restaura los registros y vuelve al programa principal.

```
// Manejo de interrupciones

#include "int16CXX.H"
#pragma origin 4  // instala la RSI en la dirección 0x0004

interrupt rsi( void)
{
    int_save_registers    // W, STATUS (and PCLATH)

    // En esta zona se debe preguntar por
    // el flag de interrupción correspondiente

    int_restore_registers // W, STATUS (and PCLATH)
}

void main()
{
   // Configurar la fuente de interrupción
   // Habilitar las interrupciones
   while(1)
    {
      // Otras tareas
    }
}
```

Programando la RSI para una interrupción externa
La idea es usar como fuente de interrupción el terminal INT del µC (PORTB.0 o también llamado RB0), para ello se configura el servidor de interrupción para tal efecto. Se usará un bit del puerto B para visualizar el evento por medio de un LED indicador. De acuerdo al esquema presentado en el ejemplo anterior:

- ☐ Se configura la fuente de interrupción, INTEDG=1
- ☐ Se activa la fuente a través de los bits INTE=1, GIE=1
- ☐ Adicionalmente se configura el PORTB.7 como salida

```
#include "int16CXX.H"
#pragma origin 4  // instala la RSI en la dirección 0x0004
```

```
interrupt rsi( void)
{
    int_save_registers    // W, STATUS (and PCLATH)

    if(INTF)
    {
      if(PORTB.7==0)PORTB.7=1;
      else PORTB.7=0;

      INTF=0; // Reset Flag
    }
    int_restore_registers // W, STATUS (and PCLATH)
}
void main()
{
 TRISB.7=0;
 PORTB.7=1;
 INTEDG=1; // Canto de subida de INT
 INTE=1;   // Habilita la Interrupción
 GIE=1;
 while(1)
    {
      // Otras tareas
    }
}
```

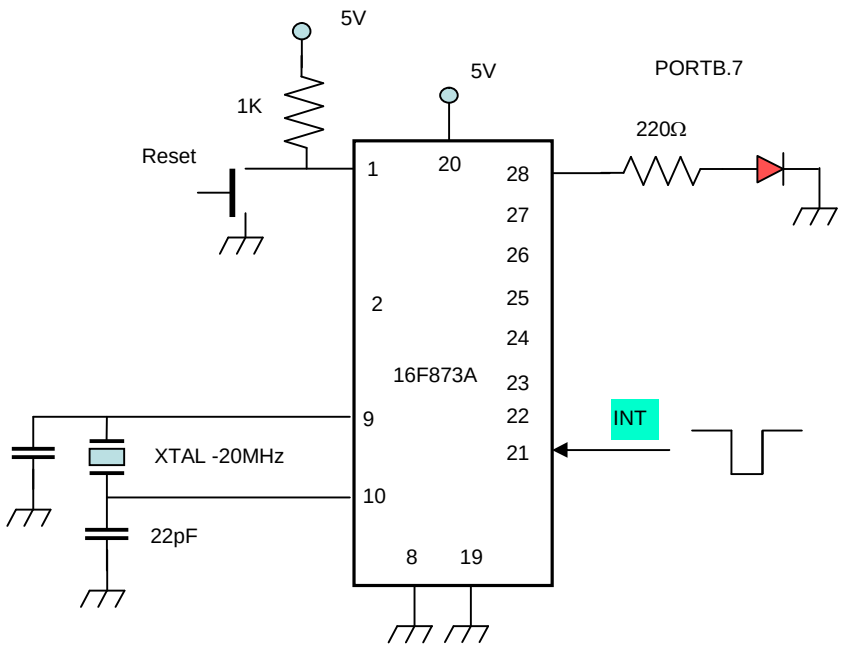


Figura 12. Manejo del terminal INT.

Cada vez que se produzca un flanco en el terminal INT, el programa salta a la RSI, verifica que el INTF=1 y cambia de estado el bit de salida. Antes de volver al programa principal, hace la variable INTF=0.

Interrupción usando Timer 0

El Timer 0, llamado por el fabricante TMR0, este módulo de 8 bits cuenta pulsos de reloj a partir de una carga inicial, cuando completa la cuenta al llegar a 0xff y pasar a 0x00, se produce un overflow. Este evento genera una interrupción. El módulo se debe reiniciar para repetir el proceso. Esta interrupción es capturada por el flag TMR0IF. En el compilador CC5X, el flag recibe el nombre de T0IF.

Las fuentes de reloj pueden ser externas, las que son ingresadas por el terminal RA4/T0CKI, o interna, la que corresponde a un factor de la frecuencia del oscilador, llamada fosc/4. Dichas fuentes y cantos se configuran

en el registro OPTION_REG. Este temporizador puede usar pre-escalares que permiten incrementar la base de tiempo. En el caso de no necesitarlos pueden ser derivados al subsistema WDT (watchdog timer).

La programación del modulo puede realizarse mediante los siguientes pasos:

- ☐ Se configura la fuente de interrupción en el registro OPTION_REG
- ☐ Se habilita el Timer 0 en el registro INTCON
- ☐ Se le da la carga inicial al TMR0, en el ejemplo será 0x00
- ☐ Se habilita la interrupción
- ☐ Adicionalmente se configura el PORTB.7 como salida

El sistema cuenta de 0x00 hasta 0xff, cuando ocurre el overflow, se activa la interrupción y salta a la RSI. En la aplicación siguiente al producirse la interrupción, se setea o resetea el bit 7 del puerto B.

La pregunta que surge en es ¿Cada cuanto tiempo se realiza este evento?, pues en la medida que se usa un temporizador, habrá tiempo involucrado. Para esto se considera, fosc/4, la cuenta inicial de TMR0 y el pre-escalar mínimo.

$$T_{\text{interrupción}} = \frac{1}{\frac{f_{osc}}{4}} \cdot (\text{pre - escalar}) \cdot (255 - \text{Carga Inicial})$$

De esta forma se tiene

$$T_{\text{interrupción}} = \frac{1}{5MHz} \cdot (2) \cdot (255 - 0) = 0.2 \cdot 10^{-6} \cdot 2 \cdot 255 = 0.102 mSeg.$$

El código siguiente propone una RSI, con el formato previamente especificado y las modificaciones respectivas para la implementación de la interrupción por el Timer 0. Cuando se produce el overflow del TMR0, se genera la interrupción haciendo que el flag de interrupción de dicho timer se hace 1. Al producirse la interrupción el control del programa salta a la RSI y pregunta por el flag, dado que es 1, entonces se ejecutan las instrucciones del if y luego se hace un reset al flag T0IF.

```
#include "int16CXX.H"
#pragma origin 4  // instala la RSI en la dirección 0x0004

interrupt rsi( void)
{
    int_save_registers    // W, STATUS (and PCLATH)

    if(T0IF)
    {
      if(PORTB.7==0)PORTB.7=1;
      else PORTB.7=0;

      TMR0=0;
      T0IF=0; // Reset Flag
    }
    int_restore_registers // W, STATUS (and PCLATH)
}

void main()
{
 TRISB.7=0;
 PORTB.7=1;

 T0CS=0;            // Fuente de clock interna
 PSA=0;             // Preescalar asociado al TMR0
 PS2=0;PS1=0;PS0=0; // 1:2

 TMR0=0;            // Carga inicial
 T0IF=0;
 TMR0IE=1;          //verificar
```

```
 GIE=1;

 while(1)
    {
      // Otras tareas
    }
}
```

Luego, el cambio de estado se podrá ver solo con un osciloscopio o sonda lógica. Es posible cambiar el pre-escalar a 1:256, esto se hace cambiando los bit PS2, PS1 y PS0. (todos los bits en 1)

$$T_{\text{interrupción}} = \frac{1}{5MHz} \cdot (256) \cdot (255 - 0) = 0.2 \cdot 10^{-6} \cdot 2 \cdot 255 \approx 13mSeg.$$

Note que si el valor de la carga inicial está más cercano a 0xff, demora menos en producirse el overflow.

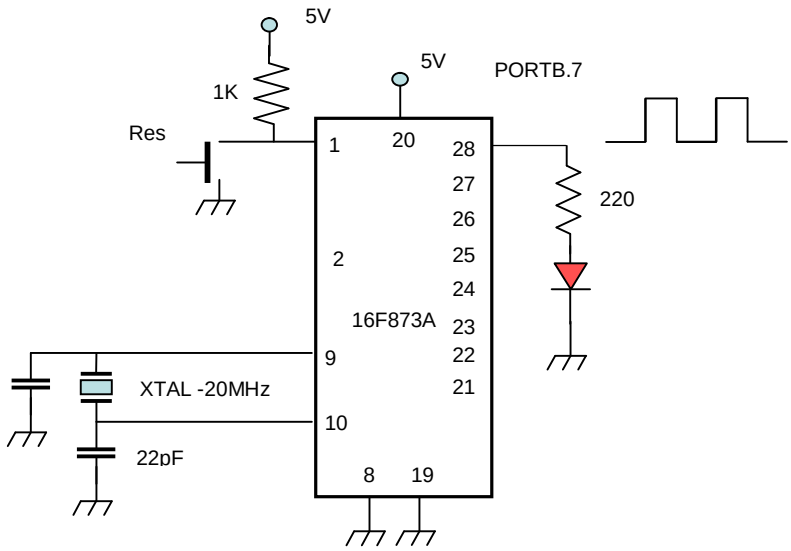


Fig. 13. Manejo Interrupción TMR0.

Usando el Timer1

El Timer1 es un módulo de Temporizador/contador de 16 bits que tiene 2 registros de lectura y escritura TMR1H y TMR1L. De esta forma se tiene que:

- El registro formado por ambos TMR1H:TMR1L se incrementa desde 0000h a FFFFh.
- Si la Interrupción TMR1 es habilitada, ésta es generada por el *overflow* el cual es capturado por el flag TMR1IF.
- Esta puede ser habilitada o deshabilitada por el TMR1IE.
- El modulo trabaja como:
  - Timer
  - Counter
  - El modo queda determinado por el bit que selecciona el clock.

- En modo **Timer**, el Timer1 se incrementa cada ciclo de instrucción.
- En modo **Counter**, el incremento es en cada canto de subida de una entrada de clock externa
- El Timer1 se habilita/deshabilita mediante el bit TMR1ON.
- También tiene una entrada de reset.

```
char contador;

#include "int16CXX.H"
#pragma origin 4

interrupt rsi( void)
{
    int_save_registers    // W, STATUS (and PCLATH)
    if (TMR1IF)  {
        TMR1 = 0;
        contador++;
        if (contador==10)
         {
         }
        TMR1IF = 0;  /* reset flag */
    }
 int_restore_registers // W, STATUS (and PCLATH)
}

void main()
{
 TMR1CS =0; // Selecciona fuente de clock
 T1CKPS0 =0;// Prescalar
 T1CKPS1= 0; // 1:1
 TMR1ON =1; // Timer ON

 TMR1=0;            // Carga inicial
 TMR1IF=0;
 TMR1IE=1;          //verificar
 GIE=1;

 while(1)
    {
      // Otras tareas
    }
}
```

Otras interrupciones

Otras fuentes de interrupciones requieren de banderas de habilitación y de detección de interrupción. Como ejemplo se tiene el Timer2 el cual requiere permite el uso del módulo CCP cuando se usa en modo PWM. De similar forma se tiene para el Timer1.

Bit	Registro	Bit Declaración (CC5x)	
TMR2IE	PIE1.1		TMR2 to PR2 Match Interrupt Enable bit
TMR1IE	PIE1.0		TMR1 Overflow Interrupt Enable bit
TMR2IF	PIR1.1		TMR2 to PR2 Match Interrupt Flag bit
TMR1IF	PIR1.0		TMR1 Overflow Interrupt Flag bit
SSPIE	PIE1.3		Synchronous Serial Port Interrupt Enable bit
SSPIF	PIR1.3		Synchronous Serial Port (SSP) Interrupt Flag bit

Programación PWM

El PWM (**P**ulse **W**itdh **M**odulation) en una señal muy utilizada para el control de motores de corriente continua. El PWM consiste en una señal cuadrada que forma un tren de pulsos, cuyo ancho puede ser variado por el usuario manteniendo el periodo original. El concepto radica en que el ancho de pulso permite variar el valor medio de la señal generado (también llamado valor de corriente continua).

Este tipo de señales puede ser generada usando los temporizadores en conjunto con las interrupciones disponibles del µC o usando el modulo CCP (Capture/Compare/PWM). Este modulo funciona en conjunto con el timer2, el proceso de interrupción involucrado es transparente para el usuario.

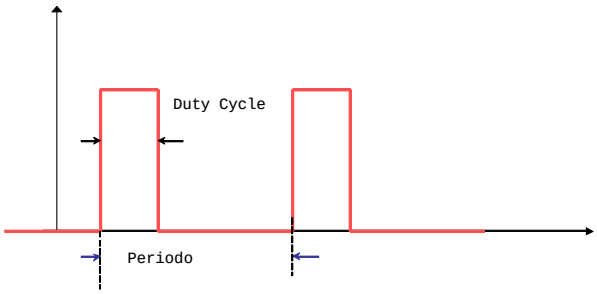


Fig. 14. Generación de PWM.

Generación de un PWM
El concepto es el siguiente, se requiere una variable que almacene la duración del periodo, una variable que almacene el ciclo de trabajo y un terminal de salida que permite visualizar la señal. Esto en conjunto con un **contador** que cuenta ciclos de reloj producido por un temporizador. Dicha cuenta se realiza en una RSI.

De esta forma, se pone el terminal de salida en 1, la base de tiempo generada por cada canto de reloj interrumpe, cada interrupción incrementa en uno el **contador**, si la cuenta es menor al ciclo de trabajo, el terminal de salida se mantiene en 1, sino baja a 0. El contador se sigue incrementando por cada interrupción, pero ahora se pregunta si la cuenta es menor o igual al largo del periodo, es decir, si la cuenta es menor que el largo del periodo, se mantiene el bit de salida en el estado actual, sino, significa que se cumplió el periodo y por lo tanto el contador se hace cero, el bit de salida se hace 1 y se repite el proceso. El ancho del pulso se maneja cambiando el valor de la variable correspondiente. Lo mismo para el periodo.

Dependiendo de la frecuencia y la resolución que se requiera del PWM, puede usar algunos de los temporizadores disponibles, ya sea el Timer 0 o el Timer 1.

Generando PWM con el TMR0
Esto se hace vía interrupción, necesitará una base de tiempo la cual será proporcionada por el TMR0. La señal de PWM saldrá por el terminal de salida PORTB.7.

Primero se debe implementar una señal de reloj cuya frecuencia se lo más grande posible, esto generará una base de tiempo muy pequeña. Así se tiene:

- PS2,PS1,PS0 con el divisor 1:2
- TMR0=254

Esta base de tiempo será el reloj principal para el PWM. El Duty Cycle más pequeño del PWM generado será un periodo de esta señal.

Se definen tres variables, el periodo del PWM, llamado TPWM, el Duty Cycle llamado DUTY y un CONTADOR que establece cuando se cumple el DUTY y cuando se cumple el periodo TPWM.

Filosofía

- Se configura la interrupción TMR0
- Se inicializa contador=0
- Se inicializa TPWM y DUTY
- Se inicializa TMR0=254
- Se define un bit de salida con el valor 1, por ejemplo PORTB.7=1
- Cada vez que interrumpe el TMR0, se ejecuta la RSI
- Cuando entra a la RSI verifica el estado del T0IF y se realizan las siguientes tareas

    - Se incrementa el CONTADOR
    - Si el CONTADOR =DUTY, se hace el PORTB.7=0
    - Si el CONTADOR=TPWM, se hace el PORTB.7=1 y contador=0
    - Se resetea T0IF=0
    - Se inicializa TMR0=254

Por ejemplo si se define TPWM=100 y DUTY=20, durante las primeras 20 interrupciones del TMR0, PORTB.7 será 1 y luego se hará 0 durante las 80 interrupciones siguientes. Al llegar el CONTADOR a 100 se hace un reset al CONTADOR y parte todo de nuevo. Para modificar el ancho del pulso se debe cambiar el valor del DUTY.

El siguiente código fuente sería una implementación, la que puede simplificarse programando los registros en forma completa y no por bits.

```
#include "int16CXX.H"
#pragma origin 4

char contador, TPWM,DUTY;

interrupt servidor_de_int( void)
{
    int_save_registers    // W, STATUS (and PCLATH)
    if(T0IF) {
              contador++;
              if(contador==DUTY) PORTB.7=0;
              if(contador==TPWM)
               {
                 PORTB.7=1;
                 contador=0;
               }
              TMR0=254;
              T0IF=0; // reset flag de int TMR0
             }
    int_restore_registers // W, STATUS (and PCLATH)
}

void main()
{
  TPWM=100;
  DUTY=20;
  TRISB.7=0; // PORTB.0 de salida
  PORTB.7=1; // Manda un 1

 //Programación del TMR0

  OPTION.5=0; // T0CK=0, fuente clock interno
  OPTION.4=0; // T0SE,  Incremento con Canto de subida
  OPTION.3=0; // PSA=0  Pre escalar asignado al TMR0
  OPTION.2=0; // PS2=0 // Pre escalar 1:2
  OPTION.1=0; // PS1=0
```

```
  OPTION.0=0; // PS0=0
  TMR0=254;    // Carga con el valor inicial 254
  INTCON.5=1;  // habilita la int por TMR0  TMR0IE=1
  GIE=1;       // Habilita todas las int
  while(1) // Loop forever
  {
  }
}
```

Programando el módulo CCP

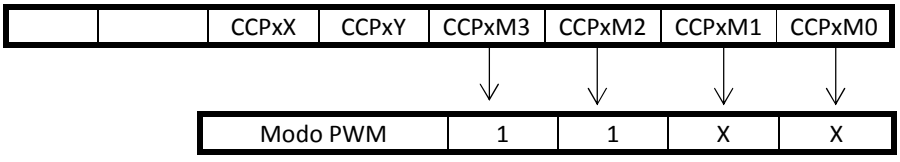
El µC tiene dos módulos, el CCP1 o CCP2 cuya salida son la 13 y 12 respectivamente. Estos módulos interactúan con el timer2. Para la configuración CCP1 se utilizan los registros TRISC, CCP1CON, CCPR1L (registros del modulo CCP1), PR2 y T2CON (registros del modulo Timer2). El PWM tiene una resolución de 1024 cuentas. Las condiciones iniciales serán el periodo del PWM y el clock del sistema.

- ☐ Se configura el modulo CCP para trabajar en modo PWM programando el registro CCP1CON.
- ☐ Se ingresa el valor codificado en 10 bits del ciclo de trabajo (Duty Cycle) en el registro CCPR1L (que es de 8 bits) y los dos bits restantes en el registro CCP1CON.4 y CCP1CON.5.
- ☐ Se configura el Timer2, a través de los registros T2CON y PR2. Esto permite definir los pre-escalares respectivos y la duración del periodo en base a la siguiente expresión lo que permite realizar los ajustes para la frecuencia requerida.

Los siguientes diagramas permiten ilustrar la programación

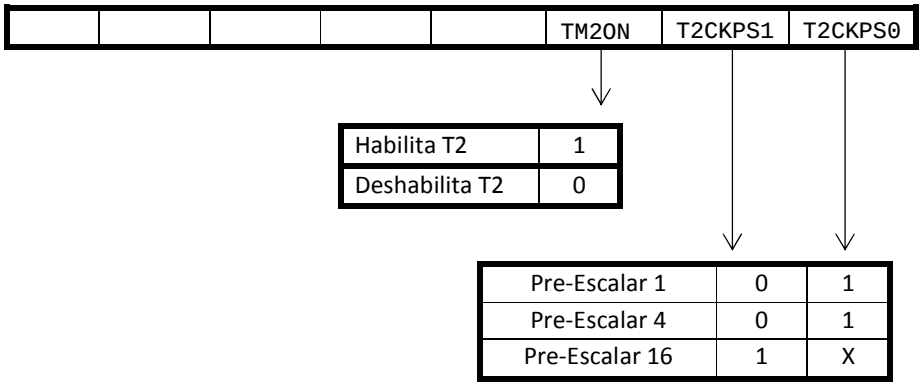
Para CCP1, se configura la palabra de control

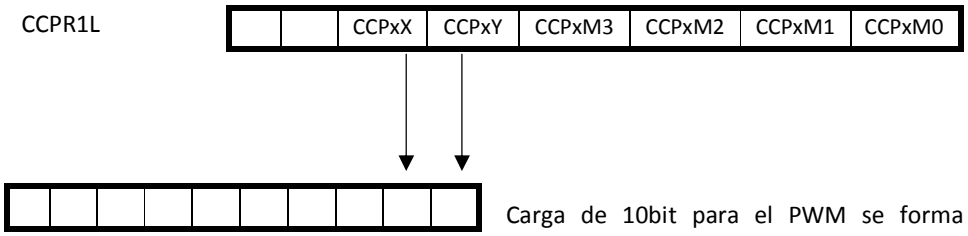
CCPCON1



PR2

T2CON





Carga de 10bit para el PWM se forma con los registros indicados

El programa siguiente muestra la configuración de los registros de control accesando en forma directa los bits respectivos.

```
// Programación de acuerdo a los bits
// Prescalar de 16 PWM Freq = 1.22kHz - Clock 20MHz

void main()
{
   TRISC.2 = 0;

 // CCP1CON=0x00001100;

   CCP1M0=0; // Modo PWM
   CCP1M1=0;
   CCP1M2=1;
   CCP1M3=1;

// Los 2 bit siguientes forma los 10bit del duty con CCPR1L
// es decir:   CCPR1L     CCP1X CCP1Y
//              1000 0000   0     0

   CCP1X=0;  // Forman 10 bit  1000 0000 00=512
   CCP1Y=0;

 // CCPR1L=0b10000000;

   CCPR1L=0x80;

  //T2CON = 0b00000110;

  T2CKPS0=0;  //Presecalar 16
  T2CKPS1=1;
  TMR2ON=1;  // Timer2 ON

  TOUTPS3=0; // Post escalares 1:1
  TOUTPS2=0;
  TOUTPS1=0;
  TOUTPS0=0;
  PR2 = 0xFF; // PERIODO Seteado a 0xFF

  while(1)
   {
     CCPR1L=0x80;
   }
 }
```

Programando el UART

Se debe programar el modulo SSP. Los terminales de transmisión y recepción están ubicados en el PORTC y corresponden al PORTC.6, para la TX y PORTC.7 para la RX. Se deben configurar los bits como entrada y salida respectivamente mediante el registro TRISC.
Como cualquier módulo, requiere de la programación de un conjunto de registros. Los métodos de TX y RX pueden ser combinados con mecanismos de interrupciones con el fin de mejorar el funcionamiento del proceso de transferencia de datos.
Los registros de configuración básicos involucrados son TRISC, SPBRG, TXSTA, RCSTA. Para la transmisión y recepción de datos se usan los registros TXREG, RCREG y los flags de interrupción para determinar cuando ha sido transmitido o recibido un dato TXIF y RCIF.

```
//Programación de RS232
```

- El registro TRISC, permite definir como entrada el bit de RX (PORTC.7) y como Salida el bit de TX (PORTC.6).
- El Registro TXSTA, configura el módulo de TX.
- El registro RCSTA configura el módulo de recepción, sin embargo, el bit RCSTA.7 =SPEN (Serial Port Enable) se debe configurar para la transmisión, por lo cual debe ser utilizado

Para evaluar la aplicación siguiente arme y conecte el circuito implementado al puerto COM del PC. Ejecute la aplicación Hyperterminal, y configúrela con 1200, 8, n, 1. Haga un reset en el µC y recibirá el mensaje "HOLA".

Se ha separado el código y se implementa una función `enviar(char c)`. Esta función recibe como argumento el carácter c, el cual es transmitido. La función es llamada por el `main()`. La Programación está más simplificada.

```
// Programa para verificación de la comunicación

void enviar(char c)
{
  while(TXIF==0) // Verifica que haya TX
    {
    }
  TXREG=c;
}

void main()
{
  TRISC.7=1; // habilita El bit para RX
  TRISC.6=0; // Habilita El bit para TX
  TXSTA=0b00100000; // Ver registro
  RCSTA.7=1; //  SPEN=1;
  SPBRG=0xFF; // De acuerdo a la tabla del manual
  while(1)
  {
   enviar('H');
   enviar('O');
   enviar('L');
   enviar('A');
   enviar(0x0d);
   enviar(0x0A);
// Puede poner una función para retardo
  }
 }
```

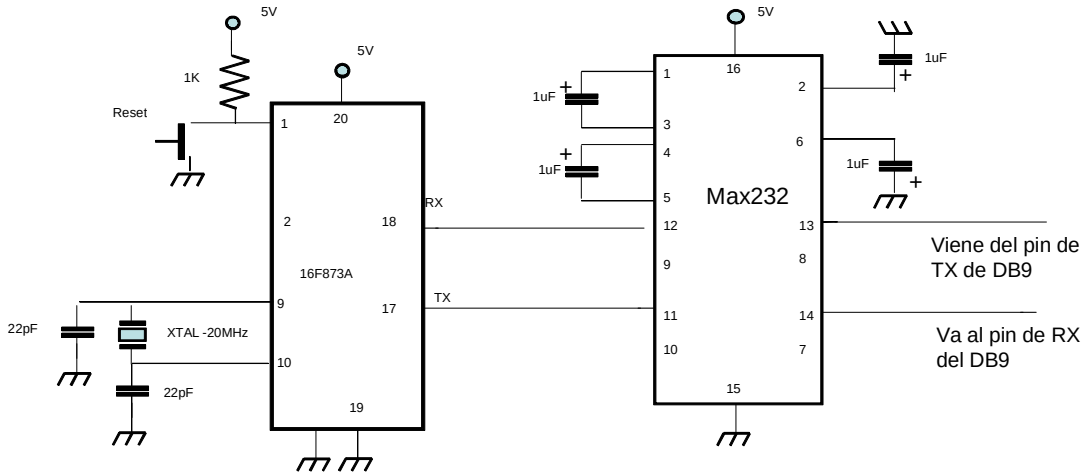


Fig. 15. Interfaz con MAX 232.

Acceso a Memoria

Pueden ser accesadas tanto la memoria EEPROM de datos como la memoria flash de programas, la primera tiene un tamaño de 256 x8 bit y la segunda 4K x 14 bit (en el caso del 16F877A, son 8Kx14bits).
La manipulación de estos periféricos permite almacenar datos para aplicaciones de configuración y aplicaciones tipo *datalogger*. En el caso de la memoria FLASH, el tamaño disponible será el tamaño que quede después de grabar la aplicación. Por lo general, las aplicaciones pueden no superar los 2Kbytes, quedando a disposición 2kbytes para el usuario.
Primero se debe configurar el acceso mediante los registros EECON1 y EECON2.
El acceso a los datos de la EEPROM es a través del registro EEADR y EEDATA, en el caso de la memoria FLASH es a través de los registros EEADR, EEADRH y EEDATA, EEDATAH, dado que el espacio de direcciones es mayor a 256 bytes y dato que el dato que es posible almacenar por dirección es de 14 bits.

El registro EECON1 está desglosado en la siguiente tabla

Bit	Registro	Bit Declaración (CC5x)	Funciones (*)
EEPGD	EECON1.7	EEPGD	Selecciona la EEPROM o la FLASH
WRERR	EECON1.3	WRERR	Error de Escritura
WREN	EECON1.2	WREN	Habilita Escritura
WR	EECON1.1	WR	Inicia Ciclo de Escritura
RD	EECON1.0	RD	Inicia Ciclo de Lectura

(*) Revisar el datasheet del µC.

Para leer la data de la EEPROM

- ☐ Se configura el EECON1
  - ☐ Escriba la dirección en el registro EEADR
  - ☐ Escriba un cero en el bit de habilitación de la memoria EEPGD=0
  - ☐ Inicie la lectura haciendo RD=1
  - ☐ Lea la data desde EEDATA

Las siguientes funciones realizan la operación de lectura y escritura en la EEPROM.

```
char leeEEPROM(char dir)
{
  char dato;
  EEADR=dir;
  EEPGD=0;
  RD=1;
  dato=EEDATA;
  Return (dato);
}

void escribeEEPROM(char dir, char dato)
{
  EEADR=dir;
  EEPGD=0;
  EEDATA= dato;
  WREN=1;
  WR=1;
}
```

Accesando la Memoria Flash
Considere el mapa de memoria de la Fig. 9. Dado que no se puede escribir desde la dirección 0x0000, ya que a partir de esa zona se encuentra la aplicación, es posible definir una zona de almacenamiento a partir de los 2Kbytes o 3KBytes. Esto es a partir de la 0x0800 o 0x0C00, por lo tanto se debe definir un offset o

desplazamiento de inicio de almacenamiento, esto para definir una dirección de inicio lógico 0x0000, que en realidad será la dirección del desplazamiento.

Secuencia

- ☐ Se configura el EECON1
- ☐ Se envía la secuencia 0x55 y 0xAA al EECON2
- ☐ Escriba un 1 en el bit de habilitación de la memoria, EEPGD=1
- ☐ Haga WR=1

El siguiente programa es un ejemplo de lo definido.

```
Uns16 leeFLASH(uns16 dir)
{
  uns16 dato;
  EEADR=dir.low8;
  EEADH=dir.high8;
  EEPGD=1;
  RD=1;
  dato.low8=EEDATA;
  dato.high8=EEDATA;

  Return (dato);
}


void escribeFLASH(char dir, uns16 dato)
{
  EEADR=dir.low8;
  EEADH=dir.high8;
  EEPGD=1;
  EECON2=0x55;
  EECON2=0xAA;

  EEDATA= dato.low8;
  EEDATAH= dato.high8;
  WREN=1;
  WR=1;
}
```

Manejo de periféricos mediante interrupciones

A continuación de plantean algunos ejemplos para el uso de interrupciones en conjunto con algunos elementos de Entrada-Salida del µC.

ADC e Interrupciones
El uso de interrupciones es fundamental para el adecuado funcionamiento del módulo ADC, sin embargo, esto resultará complicado desde el punto de vista de la programación si no se tiene claro los procesos involucrados. Un adecuado uso del módulo ADC implica que la toma de muestras debe ocurrir a intervalos regulares.

En las aplicaciones clásicas de un módulo ADC se requiere saber el tiempo de muestreo de tal forma de optimizar el almacenamiento. En este punto es vital el uso de un temporizador que controle el uso del módulo, lo que permite establecer el tiempo de muestreo. Cada TIC producido por el temporizador permitirá dar inicio al proceso de conversión. Por otro lado, para saber el momento exacto en que el dato está disponible puede ser usado el flag ADIF del módulo que indica el final del proceso.

Para iniciar este proceso básico de interrupciones, se activa la interrupción mediante el bit ADIE (**A**nalog to **D**igital **I**nterrupts **E**nable) del registro PIE1, se incorpora en el servidor de interrupciones el flag ADIF del registro PIE1, de esta forma no será necesario encuestar el bit GO/DONE para saber si la conversión ha terminando.

Se debe definir el Tiempo de muestreo, esto definirá el periodo del temporizador, dependiendo del tiempo puede usarse el TIMER0 o TIMER1. Se debe programar el módulo ADC para permitir la interrupción.
La RSI deberá cumplir atender dos interrupciones, la primera provocada por el temporizador que define el tiempo de muestreo y da el inicio al proceso de conversión del módulo ADC, haciendo GO=1, y la segundo que atiende el fin de la conversión y pone a disposición del usuario el dato. Esto ocurre cuando ADIF=1.

```
#include "int16CXX.H"
#pragma origin 4

char cont;
uns16 dato;

interrupt servidor_de_int( void)
{
     int_save_registers    // W, STATUS (and PCLATH)
     if(T0IF)
     {
       GO=1;  // Inicio conversion
       TMR0=254;
       T0IF=0; // reset flag de int TMR0
     }
     if(ADIF)  // Conversion lista
     {
       dato.high=ADRESH;
       dato.low=ADRESL;
       ADIF=0;
     }
    int_restore_registers // W, STATUS (and PCLATH)
}

void main()
{
  TRISB.7=0; // PORTB.0 de salida
  PORTB.7=1; // Manda un 1

 //Programación del TMR0

  OPTION.5=0; // T0CK=0, fuente clock interno
  OPTION.4=0; // T0SE,  Incremento con Canto de subida
  OPTION.3=0; // PSA=0  Pre escalar asignado al TMR0
```

```
  OPTION.2=0; // PS2=0 // Pre escalar 1:2
  OPTION.1=0; // PS1=0
  OPTION.0=0; // PS0=0
  TMR0=254;    // Carga con el valor inicial 254
  INTCON.5=1;  // habilita la int por TMR0  TMR0IE=1

  // Programación del ADC

  GIE=1;       // Habilita todas las int

  while(1) // Loop forever
  {
  }
}
```