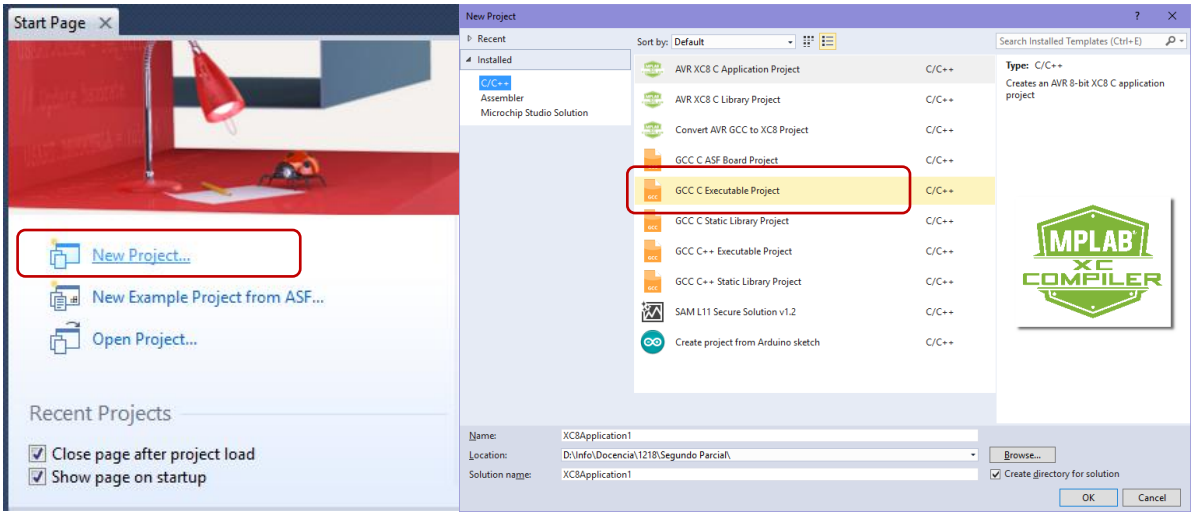
Microcontroladores

Programación en C para Micros

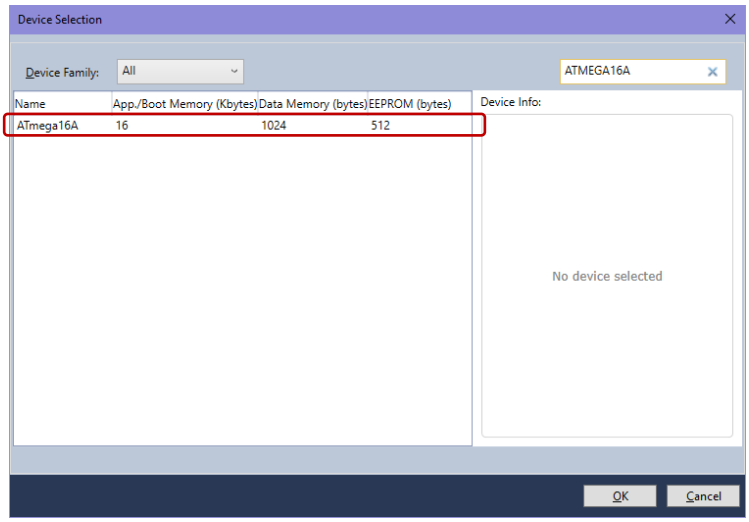
Teresa Orruñanos Guerrero

Bernardo Calabrese

Para crear un nuevo proyecto de programación del AVR ATmega16a utilizando MICROCHIP STUDIO, abra el software y cree un nuevo proyecto y en seguida elija GCC C Executable Project.



Especifique el nombre del proyecto y la carpeta en donde desea guardarlo; se sugiere elegir la opción de crear un folder para el proyecto, pues éste estará formado por varios archivos y así le será más fácil mantener organizada su información. Entonces presione el botón OK



En la siguiente ventana elija el procesador ATmega16A y presione el botón OK.

Entonces se abrirá la ventana de programación para el nuevo proyecto, donde se mostrará el siguiente código para iniciar el programa:

```
/*
 * GccApplication1.c
 *
 * Created: 03/04/2000 00:00:00 p.m.
 *  Author: María Teresa Orvañanos Guerrero
```

```
 */


#include <avr/io.h>

int main(void)
{
    while(1)
    {
        //TODO:: Please write your application code
    }
}
```

Por default el programa agrega la primera línea `#include <avr/io.h>` esto es importante pues en este "include" se especifican todas las características concretas del micro que se eligió programar, las direcciones de los puertos, etc.

También tiene un punto en donde comienza a correr el programa, éste es el main, dentro de este bloque de código antes de llegar al bloque while(1), se encontrarán definiciones de puerto y configuraciones que sólo sea necesario ejecutar una vez; casi siempre los programas del microprocesador llegan a un punto en el cual se da un ciclo infinito, este ciclo es todo aquello que se programe dentro del while(1) { }, es importante mencionar que este código se estará repitiendo infinitamente puesto que la sentencia while(1) corresponde a algo que siempre será verdadero.

En este punto vale la pena recordar que 0 es siempre el equivalente a falso, mientras que cualquier valor diferente a cero es considerado como verdadero.

Puertos

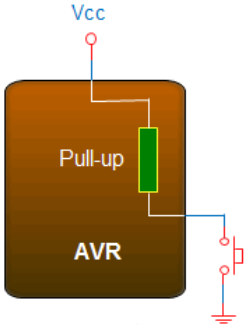
Al igual que en ensamblador, para definirnos a los puertos del microcontrolador podemos hacerlo a través de

- Port x Data Register – PORTx
- Port x Data Direction Register – DDRx
- Port x Input Pins Arddess – PINx
-

El registro DDRx se utiliza para definir un puerto como entrada o salida, así a los pines a los que se envíe 0 quedarán definidos como entradas, mientras que aquellos a los que se envíe 1 quedarán definidos como salidas.

DDRx = 0b11110000; // De esta forma queda definido como entrada en los bits bajos y salida en los bits altos

DDRx = 0b11111111; // De esta forma queda definido totalmente como salida



Cuando un pin se configura como entrada, el registro PORTx se emplea para configurar la resistencia de pull up (poniéndole un 1 al bit correspondiente). Cuando un pin se configura como salida, el registro PORTx se emplea para sacar los datos que se le pongan a esos bits.

Si el puerto está configurado como entrada entonces
PORTx = 0b11111111; // activará las resistencias de pull up internas.

Si el puerto está configurado como salida entonces
PORTx = 0b11111111; // hará que salgan 5V por todos los pines del puerto.

Por último, PINx es la forma de referirse al puerto como entrada (es decir que es el registro que se emplea para leer un puerto), por lo tanto si se desea leer lo que hay en un momento determinado en un puerto y luego guardarlo en una variable, eso se puede realizar mediante las instrucciones:

uint8_t entrada; // de esta forma se define la variable entrada de 8 bits sin signo
entrada = PINx; // guarda en la variable entra lo que hay en ese momento en el puerto x

Retardos

En C, existe una librería que permite incluir fácilmente retardos, para ello hay que incluir la línea

```
#include <util/delay.h>  //Libreria de los delays
```

Después de la que por default agregó el programa.

Una vez incluida esta librería, puede usarse en forma sencilla a través de:

```
_delay_ms(1000);   // Espera 1000ms
```

Conviene hacer notar que la función _delay_ms de la librería delay.h, tiene un límite de retardo que está relacionado con la velocidad a la que está trabajando el microcontrolador. De esta forma:

$$t_{\max en\ ms} = \frac{262.14}{F_{CPU\ (En\ MHz)}}$$

Variables y Tipos de Datos

Tabla de variables y tipos de datos del lenguaje C

Tipo de dato	Tamaño en bits	Rango de valores
char	8	0 a 255 ó -128 a 127
signed char	8	-128 a 127
unsigned char	8	0 a 255
(signed) int	16	-32,768 a 32,767
unsigned int	16	0 a 65,536
(signed) short	16	-32,768 a 32,767
unsigned short	16	0 a 65,536
(signed) long	32	-2,147,483,648 a 2,147,483,647
unsigned long	32	0 a 4,294,967,295
(signed) long long (int)	64	-2^{63} a $2^{63} - 1$
unsigned long long (int)	64	0 a 264 - 1
float	32	±1.18E-38 a ±3.39E+38
double	32	±1.18E-38 a ±3.39E+38
double	64	±2.23E-308 a ±1.79E+308

Por default el tipo double es de 32 bits en los microcontroladores sin embargo algunos compiladores como el que se emplea en este curso ofrecen la posibilidad de configurarlo para que sea de 64 bits y poder trabajar con datos más grandes y de mayor precisión.

Los especificadores signed (con signo) mostrados entre paréntesis son opcionales. Es decir, da lo mismo poner int que signed int, por ejemplo. Es una redundancia que se suele usar para “reforzar” su condición o para que se vea más ilustrativo.

El tipo char está pensado para almacenar un solo carácter ASCII como las letras. Puesto que estos datos son a fin de cuentas números también, es común usar este tipo para almacenar números de 8 bits.

En el entorno de los microcontroladores AVR, los tipos de datos int y short tienen el mismo tamaño y aceptan el mismo rango de valores, es decir, al compilador le da lo mismo si ponemos int o short. (En el lenguaje C para PC, el tipo short fue y siempre debería ser de 16 bits, en tanto que int fue concebido para adaptarse al bus de datos del procesador. Esto todavía se cumple en la programación de las computadoras, por ejemplo, un dato int es de 32 bits en un Pentium IV y es de 64 bits en un procesador Core i7).

Debido a estos tipos de variables, pueden aparecer ciertas imprecisiones en los tipos de datos que pueden perturbar la portabilidad de los programas entre los diferentes compiladores. Es por esto que el lenguaje C/C++ provee la librería **stdint.h** para definir tipos enteros que serán de un tamaño específico independientemente de los procesadores y de la plataforma software en que se trabaje. A continuación, se presentan esos tipos de variables.

Tabla de variables y tipos de datos del lenguaje C

Tipo de dato	Tamaño en bits	Rango de valores
int8_t	8	-128 a 127
uint8_t	8	0 a 255
int16_t	16	-32,768 a 32,767
uint16_t	16	0 a 65,536
int32_t	32	-2,147,483,648 a 2,147,483,647
uint32_t	32	0 a 4,294,967,295
int64_t	64	-2^{63} a $2^{63} - 1$
uint64_t	64	0 a $2^{64} - 1$

Para poder utilizar este tipo de variables, deberá incluirse el archivo stdint.h

```
#include <stdint.h>
```

Declaración de variables

Lo que se explicará a continuación, es similar a lo que se hace cuando se identifican las variables del ensamblador con la directiva .def. No se puede usar una variable si antes no se ha declarado. La forma general más simple de hacerlo es la siguiente:

Tipo_de_dato Variable;

donde Tipo_de_dato es un tipo de dato y Variable es un identificador cualquiera que se le quiere dar para referirnos a él, siempre que no sea palabra reservada.

```
uint8_t d;          // Variable para enteros de 8 bits sin signo
uint8_t b;          // Variable de 8 bits (para almacenarcaracteres ascii)
int8_t c;           // Variable para enteros de 8 bits con signo
```

```
int16_t i;          // i es una variable de 16 bits, entera, con signo
int16_t j;          // j también es una variable de 16 bits, entera, con signo
uint16_t k;         // k es una variable entera de 16 bits sin signo
```

También es posible declarar varias variables del mismo tipo, separándolas con comas.

```
float area, side;       // Declarar variables area y side de tipo float
uint8_t a, b, c;       // Declarar variables a, b y c como una variable de 8 bits sin signo
```

Variables globales y variables locales.

Las variables declaradas fuera de todas las funciones y antes de sus implementaciones tienen carácter global y podrán ser accedidas desde todas las funciones.

Las variables declaradas dentro de una función, incluyendo las variables del encabezado, tienen ámbito local. Ellas solo podrán ser accedidas desde el cuerpo de dicha función.

De este modo, puede haber dos o más variables con el mismo nombre, siempre y cuando estén en diferentes funciones. Cada variable pertenece a su función y no tiene nada que ver con las variables de otra función, por más que tengan el mismo nombre.

En la mayoría de los compiladores C para microcontroladores las variables locales deben declararse al principio de la función.

Por ejemplo, en el siguiente código hay dos variables globales (speed y limit) y cuatro variables locales, tres de las cuales se llaman count.

```
uint8_t foo(long);            // Prototipo de función

uint16_t speed;               // Variable global
const uint8_t limit = 100;    // Variable global constante

void inter(void)
{
    uint16_t count;           // Variable local
    /* Este count no tiene nada que ver con el count de las funciones main o foo */
    speed++;                  // Acceso a variable global speed
    vari = 0;                 // Esto dará ERROR porque vari solo pertenece a la función
foo. No compilará.
}
void main(void)
{
    uint16_t count;          // Variable local count
    /* Este count no tiene nada que ver con el count de las funciones inter o foo */
    count = 0;          // Acceso a count local
    speed = 0;          // Acceso a variable global speed
}
```

```
char foo(long count)    // Variable local count
    {
        uint16_t vari;          // Variable local vari
    }
```

Algo muy importante: a diferencia de las variables globales, las variables locales tienen almacenamiento temporal, es decir, se crean al ejecutarse la función y se destruyen al salir de ella.

Si dentro de una función hay una variable local con el mismo nombre que una variable global, la precedencia en dicha función la tiene la variable local. Lo más recomendable es no usar variables globales y locales con el mismo nombre.

Clases de almacenamiento

Las variables pueden ser declaradas en tres tipos de clases: auto, static y register. Cuando no se especifica, la variable queda declarada por defecto en tipo auto.

auto

Cuando se declaran este tipo de variables, éstas no se encuentran inicializadas, es por ello que el programador debe asegurarse de hacerlo antes de utilizarlas. A las variables se les asigna un espacio de memoria que se libera cuando se sale de la función en donde es empleada, esto quiere decir que los valores que tenían se perderán y no estarán ahí la próxima vez que se utilice la función

```
auto uint8_t valor;
```

```
uint8_t valor;       // al no incluir la clase, automáticamente es de tipo auto
```

static

Una variable static tiene su valor únicamente dentro de la función en que es definida (no puede usarse en otras funciones), sin embargo, se le asigna un lugar dentro de la memoria global, es decir que mantendrán el valor con el que se quedaron la última vez que se utilizó la función. Las variables tipo static son automáticamente inicializadas en 0 al inicio.

```
static uint8_t valor2;
```

register

Una variable de tipo register es similar a las variables auto, es decir que es temporal y no se inicializa en forma automática, sin embargo, la diferencia se encuentra en que las variables tipo register tratarán de ser almacenadas en los registros del microcontrolador, reduciendo de esta forma el número de ciclos que toma el acceder a una variable. Vale la pena recordar que la cantidad de registros del microcontrolador son limitadas de forma que este tipo de variables también lo son y deben ser empleadas sabiamente cuando se desee aumentar la velocidad de un proceso determinado.

```
register uint8_t valor3;
```

volatile

Cada compilador tiene la posibilidad de almacenar las variables en donde considere mejor (a menos que explícitamente se le especifique lo contrario). Es decir que una variable de tipo auto pudiera ser almacenada en un registro aún cuando no se le haya pedido directamente. Para prevenir que una variable sea almacenada en un registro se utiliza el tipo **volatile** lo cual previene que la variable cambie de valor sin desearlo.

Al utilizar interrupciones propias del microcontrolador, las variables que son utilizadas dentro del código de la interrupción deberán ser declaradas como tipo volatile, pues de otra forma sólo se modificarían dentro del código de la interrupción pero al salir seguirían conservando su valor anterior sin mostrar ningún cambio.

Por ejemplo, en el siguiente código de programa la variable count debe ser accedida desde la función interrupt como desde la función main; por eso se le declara como volatile.

```
volatile int count;     // count es variable global volátil

void interrupt(void)    // Función de interrupción
{
       // Código que accede a count
}

void main(void)         // Función principal
{
       // Código que accede a count
}
```

Casting

Puede haber muchas ocasiones en que el programador desee forzar temporalmente el tipo (tamaño) de una variable. El "casting" permite hacer que una variable declarada previamente de un tipo específico pueda ajustarse a otro tipo durante una operación determinada.

Suponga la siguiente declaración de variables:

```
int16_t x;    // variable con signo, 16-bit integer (-32768 to +32767)
int8_t  y;    // variable con signo, 8-bit character (-128 to +127)
x = 12;
```

Un ejemplo de operación con "casting" sería:

```
y = (int8_t)x;
```

El "casting" es muy importante cuando se lleva a cabo una operación entre variables de diferentes tamaños.

Considere el siguiente ejemplo:

```
uint16_t z;
uint16_t x = 150;
uint8_t  y = 63;
z = (y * 10) + x;
```

El procesamiento de esta operación, debido a que el tamaño de y es de 8 bits, al hacer la operación se asumirá que (y*10) provocará un resultado de 8 bits. Sin embargo, el resultado excede el tamaño permitido en el espacio asignado para una variable de 8 bits, por lo que el resultado quedará trunco 0x76 (118) en lugar de 0x276 (630) que era el resultado correcto. Pasará entonces a la siguiente parte de la operación, en que x es una variable de 16 bits, y entonces se le sumará 150 al 118 que se tenía, quedando en 268, lo cual es un valor equivocado.

La forma correcta de llevar a cabo esta operación sería:

z = ((uint16_t)y * 10) + x;

Por lo tanto, al escribir operaciones es muy importante pensar en términos de los máximos valores que podrían obtenerse y realizar el "casting" para ese tipo de variables.

Hay una regla: "When it doubt—cast it out."

Constantes

A la declaración de una variable se le puede añadir un especificador de tipo como const, static, auto, register, etc. En esta sección únicamente nos concentraremos en el tipo const (constante).

Una variable const debe ser inicializada en su declaración. Después de eso el compilador solo permitirá su lectura, pero no será posible cambiarle el valor.

```
const uint8_t a = 100;      // Declarar constante a
```

Sin embargo, al declarar una constante de esta forma, se ocuparán posiciones en la RAM del microcontrolador, por lo cual es mucho mejor el definir las constantes del programa con las clásicas directivas #define (como se hace en el ensamblador) ya que de esa forma no se ocupa RAM.

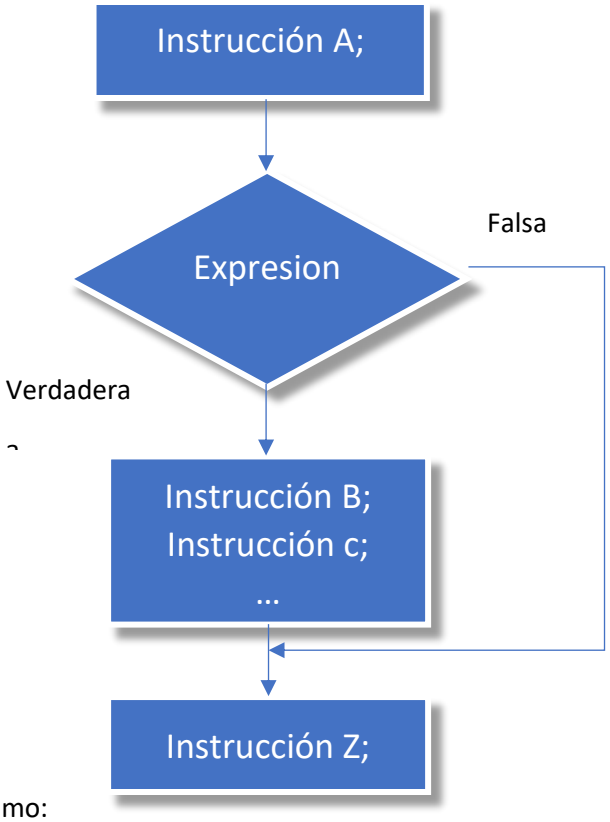
```
#define a 100           // Definir constante a
```

Sentencias de bifurcación

También son llamadas sentencias selectivas y sirven para redirigir el flujo de un programa según la evaluación de alguna condición lógica es decir permiten ejecutar una de entre varias acciones en función del valor dicha expresión.

If

La sentencia if (si condicional) hace que un programa ejecute una sentencia o un grupo de ellas si una expresión es cierta.

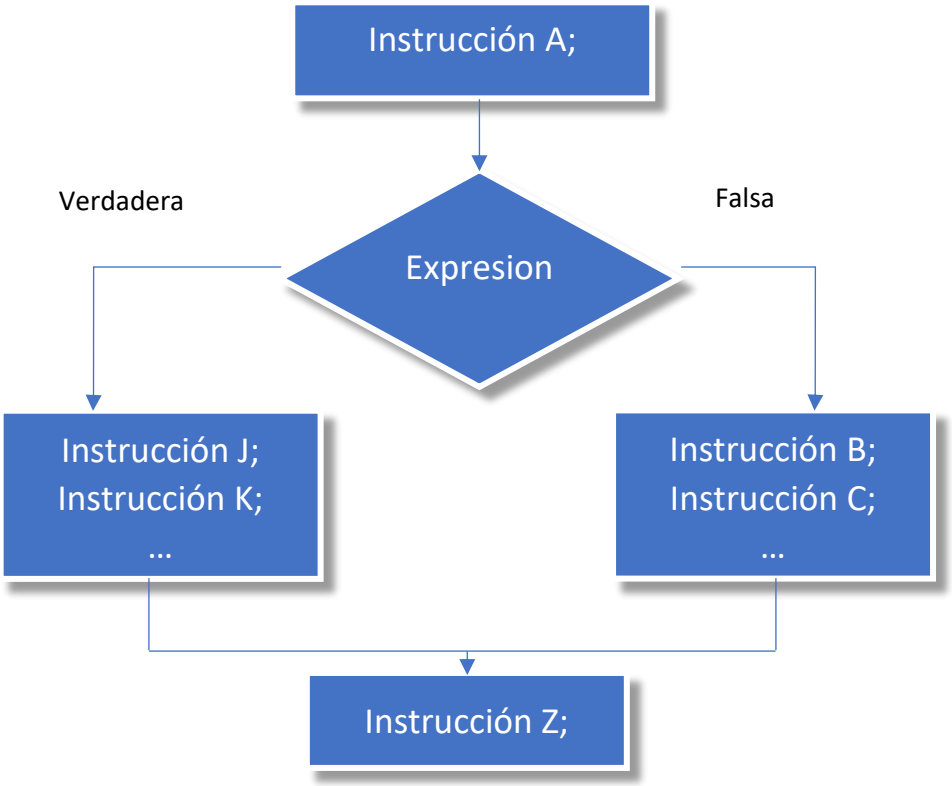


Lo cual en lenguaje C se escribe como:

```
InstruccionA;
if ( expresion )// Si expresión es verdadera, ejecutar el siguiente bloque
{                    // apertura de bloque
       InstruccionB;
       InstruccionC;
       // algunas otras sentencias
}                    // cierre de bloque
InstruccionZ;
```

if – else

La sentencia if brinda una rama que se ejecuta cuando una condición lógica es verdadera. Cuando el programa requiera dos ramas, una que se ejecute si cierta expresión es cierta y otra si es falsa, entonces se debe utilizar la sentecia if – else.



Lo cual en lenguaje C se escribe como:

```
   InstruccionA
if ( expresion )        // Si expression es verdadera, ejecutar
{                       // este bloque
       InstruccionB;
       InstruccionC;
       // ...
}
else                    // En caso contrario, ejecutar este bloque
{
       InstruccionJ;
       InstruccionK;
       // ...
}
InstruccionZ;
// ...
```

if – else – if escalonada

Es la versión ampliada de la sentencia if – else.

En el siguiente código se comprueban tres condiciones lógicas, aunque podría haber más. Del mismo modo, se han puesto dos instrucciones por bloque solo para simplificar el esquema.

```
    if ( expresion1 )     // Si expresion1 es verdadera ejecutar este bloque
    {
           Instruccion1;
           Instruccion2;
    }
    else if ( expresion2 )    // En caso contrario y si expresion2 es verdadera, ejecutar este
bloque
    {
              Instruccion3;
           Instruccion4;
    }
    else if ( expresion3 )    // En caso contrario y si expresion3 es verdadera, ejecutar este
bloque
    {
              Instruccion5;
           Instruccion6;
    }
    else                        // En caso contrario, ejecutar este bloque
    {
           Instruccion7;
           Instruccion8;
    };
    // ...
```

Las “expresiones” se evalúan de arriba a abajo. Cuando alguna de ellas sea verdadera, se ejecutará su bloque correspondiente y los demás bloques serán saltados. El bloque final (de else) se ejecuta si ninguna de las expresiones es verdadera. Además, si dicho bloque está vacío, puede ser omitido junto con su else.

switch

La sentencia switch puede ser considerada como una forma más estructurada de la sentencia if – else – if escalonada. Para elaborar el código en C se usan las palabras reservadas switch, case, break y default.

El siguiente esquema presenta tres case’s pero podría haber más, así como cada bloque también podría tener más instrucciones.

```
    switch ( expresion )
    {
           case constante1:  // Si expresion = constante1, ejecutar este bloque
              instruccion1;
              instruccion2;
              break;
           case constante2:  // Si expresion = constante2, ejecutar este bloque
              instruccion3;
              instruccion4;
              break;
           case constante3:  // Si expresion = constante3, ejecutar este bloque
              instruccion5;
```

```
            instruccion6;
            break;
        default:        // Si expression no fue igual a ninguna de las constantes anteriores,
ejecutar este bloque
            instruccion7;
            instruccion8;
            break;
    }
    instruccionX;
    // ...
```

Donde constante1, constante2 y constante3 deben ser constantes enteras, por ejemplo, 2, 0x45, 'a', etc. ('a' tiene su correspondiente código ASCII 165, que es, a fin de cuentas, un entero.)

El programa solo ejecutará uno de los bloques dependiendo de qué constante coincida con la expresión. Usualmente los bloques van limitados por llaves, pero en este caso son opcionales, dado que se pueden distinguir fácilmente. Los bloques incluyen la sentencia break que hace que el programa salga del bloque de switch y ejecute la sentencia que sigue (en el código: sentenciaX). ¡Importante!: de no poner break, también se ejecutará el bloque del siguiente case, sin importar si su constante coincida con la expresión o no.

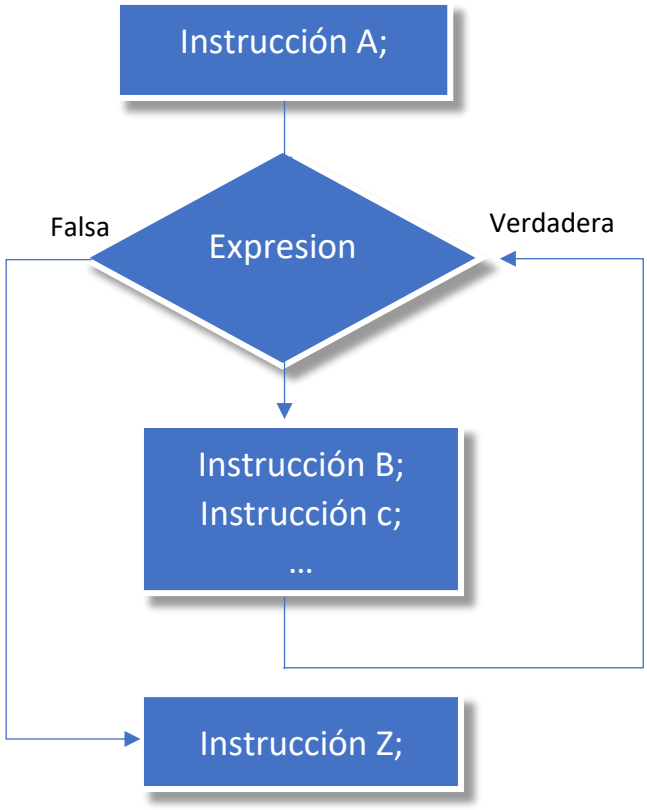
No es necesario poner el default si este bloque está vacío.

Sentencias iterativas

Las sentencias iterativas sirven para que el programa ejecute una sentencia o un grupo de ellas un número determinado o indeterminado de veces.

while

El cuerpo o bloque de este bucle se ejecutará una y otra vez mientras (while) la expresión sea verdadera.



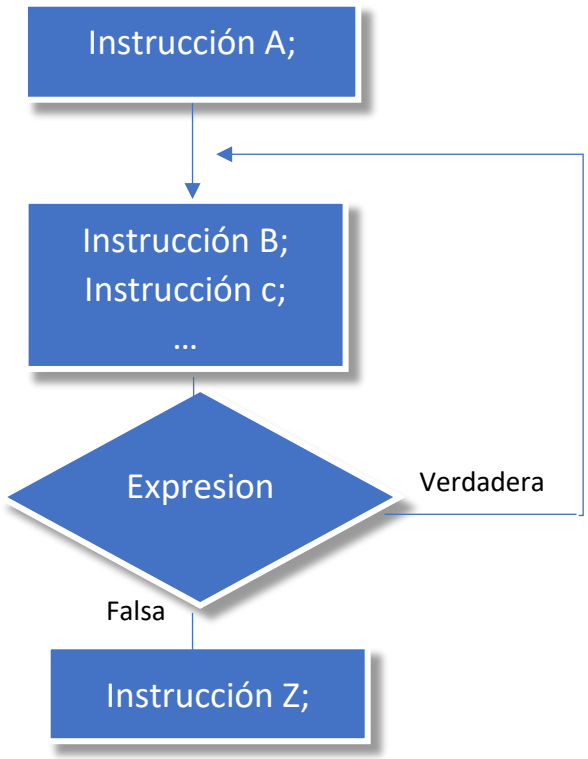
El bucle while en C se lee así: mientras (while) expresión sea verdadera, ejecutar el siguiente bloque.

```
instruccionA;
while ( expresion )    // Mientras expression sea verdadera, ejecutar el siguiente bloque
{
        instruccionB;
        instruccionC;
        // ...
}
instruccionZ;
// ...
```

Se debe notar que en este caso primero se evalúa la expresión. Por lo tanto, si desde el principio la expresión es falsa, el bloque de while no se ejecutará nunca. Por otro lado, si la expresión no deja de ser verdadera, el programa se quedará dando vueltas “para siempre”.

do – while

Es una variación de la sentencia while simple. La principal diferencia es que la condición lógica (expresión) de este bucle se presenta al final. Como se ve en la siguiente figura, esto implica que el cuerpo o bloque de este bucle se ejecutará al menos una vez.



La sentencia do – while se lee: Ejecutar (do) el siguiente bloque, mientras (while) expresión sea verdadera.

```
instruccionA;
do
{
	instruccionB;
	instruccionC;
	// ...
} while ( expresion );
instruccionZ;
// ...
```

for

Las dos sentencias while y do – while, se suelen emplear cuando no se sabe de antemano la cantidad de veces que se va a ejecutar el bucle. En los casos donde el bucle involucra alguna forma de conteo finito es preferible emplear la sentencia for.

Ésta es la sintaxis general de la sentencia for en C:

```
for ( expresion_1 ; expresion_2 ; expresion_3 )
{
	instruccion1;
	instruccion2;
	// ...
}
```

Funciona de la siguiente forma:

expresion_1 suele ser una sentencia de inicialización, desde donde se va a empezar a contar.

expresion_2 se evalúa como condición lógica para que se ejecute el bloque, en que momento va a parar el ciclo.

expression_3 es una sentencia que controla a expresion_2, es decir cómo se va a ir incrementado la cuenta después de cada iteración.

Por ejemplo, si i es una variable

```
for ( i = 0 ; i < 10 ; i++ )
{
	instrucciones;
}
```

Se lee: para (for) i igual a 0 hasta que sea menor que 10 ejecutar sentencias. La sentencia i++ indica que i se incrementa en uno tras cada ciclo. Así, el bloque de for se ejecutará 10 veces, cuando i se incremente desde 0 hasta 9.

En este otro ejemplo las sentencias se ejecutan desde que i valga 10 hasta que valga 20. Es decir, el bucle dará 11 vueltas en total.

```
for ( i = 10 ; i <= 20 ; i++ )
{
	instrucciones;
}
```

El siguiente bucle for empieza con i inicializado a 100 y su bloque se ejecutará mientras i sea mayor o igual a 0. Por supuesto, en este caso i se decrementa tras cada ciclo.

```
for ( i = 100 ; i >= 0 ; i-- )
{
	instrucciones;
}
```

Operadores

Sirven para realizar operaciones aritméticas, lógicas, comparativas, etc. Según esa función se clasifican en los siguientes grupos.

Operadores aritméticos

Además de los típicos operadores de suma, resta, multiplicación y división, están los operadores de módulo, incremento y decremento.

Operador	Acción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo. Retorna el residuo de una división entera. Solo se debe usar con números enteros.
++	Incrementar en uno
--	Decrementar en uno

Ejemplos:

```
uint8_t a, b, c;              // Declarar variables a, b y c
c = a + b;                    // Sumar a y b. Almacenar resultado en c
b = b * c;                    // Multiplicar b por c. Resultado en b
b = a / c;                    // Dividir a entre c. Colocar resultado en b
a = a + c - b;                // Sumar a y c y restarle b. Resultado en a
c = (a + b) / c;              // Dividir a+b entre c. Resultado en c
b = a + b / c + b * b;        // Sumar a más b/c más b×b. Resultado en b
c = a % b;                    // Residuo de dividir a÷b a c
a++;                          // Incrementar a en 1
b--;                          // Decrementar b en 1
++c;                          // Incrementar c en 1
--b;                          // Decrementar b en 1
```

Si ++ o -- están antes del operando, primero se suma o resta 1 al operando y luego se evalúa la expresión.

Si ++ o -- están después del operando, primero se evalúa la expresión y luego se suma o resta 1 al operando.

```
uint8_t a, b;                         // Declarar variables enteras a y b

a = b++;                // Lo mismo que a = b; y luego b = b + 1;
a = ++b;                // Lo mismo que b = b + 1; y luego a = b;

if (a++ < 10)           // Primero comprueba si a < 10 y luego incrementa a en 1
{
      // algún código
}

if (++a < 10)           // Primero incrementa a en 1 y luego comprueba si a < 10
{
      // algún código
}
```

Operadores de bits

Se aplican a operaciones lógicas con variables a nivel binario. Si bien son operaciones que producen resultados análogos a los de las instrucciones de ensamblador los operadores lógicos del C pueden operar sobre variables de distintos tamaños, ya sean de 1, 8, 16 ó 32 bits.

Operador	Acción
&	AND a nivel de bits
|	OR nivel de bits
^	XOR (OR exclusiva) a nivel de bits
~	Complemento a uno a nivel de bits
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha

```
uint8_t m;              // variable de 8 bits
uint16_t n;               // variable de 16 bits
m = 0x48;               // m será 0x48
m = m & 0x0F;           // Después de esto m será 0x08
m = m | 0x24;           // Después de esto m será 0x2F
m = m & 0b11110000;     // Después de esto m será 0x20
n = 0xFF00;             // n será 0xFF00
n = ~n;                 // n será 0x00FF
m = m | 0b10000001;     // Setear bits 0 y 7 de variable m
m = m & 0xF0;           // Limpiar nibble bajo de variable m
m = m ^ 0b00110000;     // Invertir bits 4 y 5 de variable m
m = 0b00011000;         // Cargar m con 0b00011000
m = m >> 2;             // Desplazar m 2 posiciones a la derecha, ahora m será 0b00000110
n = 0xFF1F;
n = n << 12;            // Desplazar n 12 posiciones a la izquierda, ahora n será 0xF000;
m = m << 8;             // Después de esto m será 0x00
```

Cuando una variable se desplaza hacia un lado, los bits que salen por allí se pierden y los bits que entran por el otro lado son siempre ceros. Es por esto que, en la última sentencia, m = m << 8, el resultado es 0x00. En el lenguaje C no existen operadores de rotación.

Operadores relacionales

Se emplean para construir las condiciones lógicas de las sentencias de control. La siguiente tabla muestra los operadores relacionales disponibles.

Operador	Acción
==	Igual
!=	Diferente (no igual)
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Operadores lógicos

Generalmente se utilizan para enlazar dos o más condiciones lógicas simples.

Operador	Acción
&&	AND lógica
||	OR lógica
!	Negación lógica

```
if( !(a==0) )           // Si a igual 0 sea falso
{
       // instrucciones
}

if( (a<b) && (a>c) )    // Si a<b y a>c son verdaderas
{
       // instrucciones
}

while( (a==0) || (b==0) )   // Mientras a sea 0 ó b sea 0
{
       // instrucciones
}
```

Composición de operadores

Se utiliza en las operaciones de asignación y permite escribir código más abreviado.

Es importante resaltar que no debe haber ningún espacio entre el operador y el signo igual.

```
uint16_t a;                 // Declarar a
a += 50;                // Es lo mismo que a = a + 50;
a += 20;                // También significa sumarle 20 a a
a *= 2;                 // Es lo mismo que a = a * 2;
a &= 0xF0;              // Es lo mismo que a = a & 0xF0;
a <<= 1;                // Es lo mismo que a = a << 1;
```

Precedencia de operadores

Una expresión puede contener varios operadores, de esta forma:

```
b = a * b + c / b;      // a, b y c son variables
```

En esta sentencia no queda claro en qué orden se ejecutarán las operaciones indicadas. Hay ciertas reglas que establecen dichas prioridades; por ejemplo, las multiplicaciones y divisiones siempre se ejecutan antes que las sumas y restas. Pero es más práctico emplear los paréntesis, los cuales ordenan que primero se ejecuten las operaciones de los paréntesis más internos.

Por ejemplo, las tres siguientes sentencias son diferentes.

```
b = (a * b) + (c / b);
b = a * (b + (c / b));
b = ((a * b) + c)/ b);
```

También se pueden construir expresiones condicionales, así:

```
if ( (a > b) && ( b < c) )  // Si a>b  y  b<c, ...
{
      // ...
}
```

Funciones

Una función es un bloque de instrucciones identificado por un nombre; puede recibir y devolver datos. En bajo nivel, en general, las funciones operan como las subrutinas de ensamblador, es decir, al ser llamadas, se guarda en la Pila el valor actual del PC (Program Counter), después se ejecuta todo el código de la función y finalmente se recobra el PC para regresar de la función.

A continuación, se representa la forma general de una función:

```
tipo_de_dato1 Nombre_de_funcion(tipo_de_dato2 argumento1, tipo_de_dato3 argumento2, ... )
{
      // Cuerpo de la función
      // ...
      return Dato_de_regreso;   // Necesario solo si la función retorna algún valor
}
```

Donde:

El Nombre_de_funcion puede ser un identificador cualquiera.

tipo_de_dato1 es un tipo de dato que identifica el parámetro de salida; se utiliza únicamente cuando la función regresará un valor, en caso de que se desee programar una función que no regrese ningún valor se debe poner la palabra reservada void (vacío, en inglés).

argumento1 y argumento2 (puede haber más) son las variables de tipos tipo_de_dato2, tipo_de_dato3..., que recibirán los datos que se le pasen a la función. Si no hay ningún parámetro de entrada, se pueden dejar los paréntesis vacíos o escribir un void entre ellos.

Funciones sin parámetros

Para una función que no recibe ni devuelve ningún valor, el código anterior se reduce al siguiente esquema:

```
void Nombre_de_funcion( void )
{
    // Cuerpo de la función
}
```

Y se llama escribiendo su nombre seguido de paréntesis vacíos, así:

```
Nombre_de_funcion();
```

La función principal main es otro ejemplo de función sin parámetros. Dondequiera que se ubique, siempre debería ser la primera en ejecutarse; de hecho, no debería terminar.

```
void main (void)
{
    // Cuerpo de la función
}
```

Funciones con parámetros por valor

El valor devuelto por una función se indica con la palabra reservada return.

Para llamar a una función con parámetros es importante respetar el orden y el tipo de los parámetros que ella recibe. El primer valor pasado corresponde al primer parámetro de entrada; el segundo valor, al segundo parámetro; y así sucesivamente si hubiera más.

Cuando una variable es entregada a una función, en realidad se le entrega una copia suya. De este modo, el valor de la variable original no será alterado.

```
uint8_t minor ( uint8_t arg1, uint8_t arg2, uint8_t arg3 )
{
       uint8_t min;                    // Declarar variable min
       min = arg1;             // Asumir que el menor es arg1

       if ( arg2 < min )       // Si arg2 es menor que min
       min = arg2;             // Cambiar a arg2
```

```
        if ( arg3 < min )       // Si arg3 es menor que min
        min = arg3;             // Cambiar a arg3

        return min;             // Regresar valor de min
}

void main (void)
{
        uint8_t a, b, c, d;           // Declarar variables a, b, c y d

        /* Aquí asignamos algunos valores iniciales a 'a', 'b' y 'c' */
        /* ... */

        d = minor(a,b,c);       // Llamar a minor
        // En este punto 'd' debería ser el menor entre 'a', 'b' y 'c'
        while (1);              // Bucle infinito
}
```

En el programa mostrado la función minor recibe tres parámetros de tipo int y devuelve uno, también de tipo int, que será el menor de los números recibidos.

Al llamar a minor el valor de a se copiará a la variable arg1; el valor de b, a arg2 y el valor de c, a arg3. Después de ejecutarse el código de la función el valor de retorno (min en este caso) será copiado a una variable temporal y de allí pasará a d.

Funciones con parámetros por referencia

La función que recibe un parámetro por referencia puede cambiar el valor de la variable pasada. La forma clásica de estos parámetros se puede identificar por el uso del símbolo &, tal como se ve en el siguiente código:

```
int minor ( int & arg1, int & arg2, int & arg3 )
{
        // Cuerpo de la función.
        // arg1, arg2 y arg3 son parámetros por referencia.
        // Cualquier cambio hecho a ellos desde aquí afectará a las variables
        // que fueron entregadas a esta función al ser llamada.
}
```

En el código anterior, no es necesario indicar que se pasara la referencia de la variable, y esta función se manda a llamar de manera normal. Sin embargo, hay otra forma de pasar parámetros por referencia

```
int minor ( int *arg1, int *arg2, int *arg3 )
{
        int min;                // Declarar variable min
        min = *arg1;            // Asumir que el menor es arg1, es necesario agregar el
asterisco

        if ( arg2 < min )
        min = *arg2;
```

```
        if ( arg3 < min )
        min = *arg3;

        return min;            // Regresar valor de min, los valores a , b, c han sido
afectados

    }
void main (void)
    {
        int a, b, c, d;        // Declarar variables a, b, c y d

        /* Aquí asignamos algunos valores iniciales a 'a', 'b' y 'c' */
        /* ... */

        d = minor(&a,&b,&c);   // Llamar a minor, los valores de a b y seran afectados, y es
                               necesario enviarlos con un &
        while (1);             // Bucle infinito
    }
```

Prototipos de funciones

El prototipo de una función le informa al compilador las características que tiene, como su tipo de retorno, el número de parámetros que espera recibir, el tipo y orden de dichos parámetros. Por eso se deben declarar al inicio del programa.

El prototipo de una función es muy parecido a su encabezado, se pueden diferenciar tan solo por terminar en un punto y coma (;). Los nombres de las variables de entrada son opcionales.

Por ejemplo, en el siguiente boceto de programa los prototipos de las funciones main, func1 y func2 declaradas al inicio del archivo permitirán que dichas funciones sean accedidas desde cualquier parte del programa. Además, sin importar dónde se ubique la función main, ella siempre será la primera en ejecutarse. Por eso su prototipo de función es opcional (si se desea puede no incluirse en las declaraciones al comienzo)

```
    #include <avr.h>

    void func1(char m, long p); // Prototipo de función "func1"
    uint8_t func2(int a);          // Prototipo de función "func2"
    void main(void);            // Prototipo de función "main". Es opcional

    void main(void)
    {
        // Cuerpo de la función
        // Desde aquí se puede acceder a func1 y func2
    }
    void func1(char m, long p)
    {
        // Cuerpo de la función
        // Desde aquí se puede acceder a func2 y main
    }
```

```
uint8_t func2(int a)
{
	// Cuerpo de la función
	// Desde aquí se puede acceder a func1 y main
}
```

Si las funciones no tienen prototipos, el acceso a ellas será restringido. El compilador solo verá las funciones que están implementadas encima de la función llamadora o, de lo contrario, mostrará errores de "función no definida".

Manejo de puertos en forma individual

En ocasiones es necesario consultar o escribir en determinados pines o bits en forma individual sin afectar a los demás bits.

Por ejemplo, si se desea activar el bit 0 del puerto D como pin de salida, se puede mandar la instrucción:

DDRD = 0b00000001;

Sin embargo con esta instrucción se afectan todos los bits; si lo que se desea es modificar únicamente un bit entonces podría escribirse:

DDRD = DDRD | 0b00000001;

Otra forma de enviar esta instrucción sería:

DDRD |= (1<<0); //Provoca que el pin 0 de D sea salida ; es lo mismo que DDRD = DDRD | 0b00000001

En ambas instrucciones lo que se indica es que el nuevo valor de DDRD será igual a la operación lógica OR entre el valor anterior de DDRD y 0b00000001.

De forma similar estas instrucciones pueden emplearse con PORTD, por ejemplo

PORTD |= (1<<0); // El bit 0 del puerto D es 1, el resto no cambia; es lo mismo que PORTD = PORTD | 0b00000001

Ahora, si lo que se desea es que un pin específico cambie su valor a 0, puede emplearse la siguiente expresión

DDRD &= ~(1<<0); /Provoca que el pin0 de D sea entrada ; es lo mismo que DDRD = DDRD & 0b11111110

O bien también podría usarse con:

PORTD &= ~(1<<0); //El puerto D 0 saca 0, el resto no cambia; es lo mismo que PORTD = PORTD & 0b11111110

Esto sucede debido a que ~(1<<0) = 0b11111110 y se realiza una operación AND entre PORTD y 0b11111110 en la cual todos los bits más significativos conservan su valor original, y solamente el bit menos significativo cambia a 0.

Las instrucciones lógicas que pueden emplearse de esta forma son:

& AND	| OR	^ XOR	~ NOT
0&0=0	0|0=0	0^0=0	~0=1
0&1=0	0|1=1	0^1=1	~1=0
1&0=0	1|0=1	1^0=1	
1&1=1	1|1=1	1^1=0	

EJERCICIO

Analice el siguiente código

```
#include <avr/io.h>
#include <util/delay.h>

uint8_t main(void)
{
	DDRD |= (1<<0);			//

	while(1)
	{
		PORTD ^= (1<<0);	//
		_delay_ms(100);
	}
}
```

EJERCICIO

Analice el siguiente código

```
#define F_CPU 1000000                    //Fijamos la velocidad del CPU a 1MHz

#include <avr/io.h>                      //Libreria de entrada y salida
#include <util/delay.h>                  //Libreria de los delays

int main(void)
{
	DDRB |= (1<<1);                  // DDB1=1=salida
	DDRB &= ~(1<<0);                 // DDB0=0=entrada

	PORTB |= (1<<0);                 // pull-up activa para PB0
	PORTB &= ~(1<<1);                // inicia con el led apagado

	while(1)
	{
		if (!(PINB & (1<<0)))    //Si el pin de entrada (pin0) tiene un 0…
		PORTB |= (1<<1);         //enciende el led
		else
		PORTB &= ~(1<<1);        //apaga el led
	}
}
```

FUNCIONES ÚTILES

```
uint8_t cero_en_bit(volatile uint8_t *LUGAR, uint8_t BIT)
{
	return (!(*LUGAR&(1<<BIT)));
}

uint8_t uno_en_bit(volatile uint8_t *LUGAR, uint8_t BIT)
{
	return (*LUGAR&(1<<BIT));
}
```

¿CÓMO SE USAN LAS FUNCIONES ÚTILES?

```
if (cero_en_bit(&PINA,0))
		{
			//si está el pin 0 tiene un cero hace esto…
			//código al presionar
			_delay_ms(50);
			while(cero_en_bit(&PINA,0)){}  //esta es una traba mientras tiene 0
			_delay_ms(50);
			//código al soltar

		}
```

Funciones de Interrupción

La Función de Interrupción o ISR va siempre identificada por su Vector de Interrupción, y su esquema varía ligeramente entre un compilador y otro, puesto que no existe en el lenguaje C un formato estándar. Lo único seguro es que es una función que no puede recibir ni devolver ningún parámetro.

En el compilador AVR GCC (WinAVR) la función de interrupción se escribe con la palabra reservada ISR acompañada del Vector_de_Interrupcion.

El Vector_de_Interrupcion tiene exactamente el mismo nombre que el indicado en el datasheet del ATmega16A pero con la terminación _vect . Las definiciones según el archivo son las siguientes:

```
/* Interrupt vectors */
/* External Interrupt 0 */
#define INT0_vect                   _VECTOR(1)
/* External Interrupt 1 */
#define INT1_vect                   _VECTOR(2)
/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect            _VECTOR(3)
/* Timer/Counter2 Overflow */
#define TIMER2_OVF_vect             _VECTOR(4)
/* Timer/Counter1 Capture Event */
#define TIMER1_CAPT_vect            _VECTOR(5)
/* Timer/Counter1 Compare Match A */
#define TIMER1_COMPA_vect           _VECTOR(6)
/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect           _VECTOR(7)
/* Timer/Counter1 Overflow */
#define TIMER1_OVF_vect             _VECTOR(8)
/* Timer/Counter0 Overflow */
#define TIMER0_OVF_vect             _VECTOR(9)
/* SPI Serial Transfer Complete */
#define SPI_STC_vect                _VECTOR(10)
/* USART, RX Complete */
#define USART_RX_vect               _VECTOR(11)
/* USART Data Register Empty */
#define USART_UDRE_vect             _VECTOR(12)
/* USART, TX Complete */
#define USART_TX_vect               _VECTOR(13)
/* ADC Conversion Complete */
#define ADC_vect                    _VECTOR(14)
/* EEPROM Ready */
#define EE_RDY_vect                 _VECTOR(15)
/* Analog Comparator */
#define ANA_COMP_vect               _VECTOR(16)
/* Two-wire Serial Interface */
#define TWI_vect                    _VECTOR(17)
/* External Interrupt Request 2 */
#define INT2_vect                   _VECTOR(18)
/* TimerCounter0 Compare Match */
#define TIMER0_COMP_vect            _VECTOR(19)
/* Store Program Memory Read */
#define SPM_RDY_vect                _VECTOR(20)
```

De ahí que una vez configurada una interrupción, el código correspondiente se debe introducir en:

```
ISR (Vector_de_Interrupcion)
{
    // Código de la función de interrupción.
    // No requiere limpiar el flag respectivo. El flag se limpia por hardware
}
```

Control de las Interrupciones

Cada interrupción habilitada y disparada, saltará a su correspondiente Función de Interrupción o ISR, de modo que a diferencia de algunos otros microcontroladores no será necesario sondear los flags de interrupción para conocer la fuente de interrupción.

Los AVR tienen un hardware que limpia automáticamente el bit de Flag apenas se empiece a ejecutar la función de interrupción. Pero puesto que los flags se habilitan independientemente de si las interrupciones están habilitadas o no, en ocasiones será necesario limpiarlos por software y en ese caso debemos tener la especial consideración de hacerlo escribiendo un uno y no un cero en su bit respectivo.

Al ejecutarse la función de interrupción también se limpia por hardware el bit enable general I (el que se habilita con la instrucción sei();)para evitar que se disparen otras interrupciones cuando se esté ejecutando la interrupción actual. Sin embargo, la arquitectura de los AVR le permite soportar ese tipo de interrupciones, llamadas recurrentes o anidadas, y si así lo deseamos podemos setear en el bit I dentro de la ISR actual.

Este bit de enable general para las interrupciones, es de especial importancia, es por ello existen dos exclusivas instrucciones de ensamblador llamadas sei (para setear I) y cli (para limpiar I). En C las instrucciones correspondientes son:

```
sei();
cli();
```

Para poder emplearlas es necesario agregar la siguiente librería:

```
#include <avr/interrupt.h> //Libreria de interrupciones
```

UNIVERSIDAD PANAMERICANA

- A todos los alumnos que han llevado mi curso de microcontroladores a lo largo ya de varios años, ustedes han sido la motivación para realizar este manual, buscando con el facilitar la adquisición de conocimientos e incentivar su capacidad de investigación para adquirir aún más. De no ser por ustedes no hubiese sido posible la realización de este manual.

- A mi mamá y a mi hija, quien siempre ha sido un gran apoyo para mí al motivarme e impulsarme para buscar siempre crecer tanto a nivel personal como profesional.

- Al Dr. Enrique Arámbula, quien fue mi profesor de microcontroladores al estudiar la carrera de Ingeniería en Electrónica y Sistemas Digitales, gracias por formar en mí el gusto por la electrónica digital y por la investigación.

- A la Universidad Panamericana, campus Aguascalientes, por haber confiado en mí desde el inicio y por brindarme la posibilidad de trabajar con ellos en esta labor que para mí resulta apasionante.

- A mis colegas en la universidad por sus consejos y amistad, gracias por su invaluable apoyo.

María Teresa Orvañanos Guerrero