Tutorial AVR desde 0

Introducción a la programación
de microcontroladores AVR

Felixls
http://sergiols.blogspot.com

Licencia

Prefacio

Este tutorial es una introducción a la programación de microcontroladores Atmel AVR, en particular usando el lenguaje C y el compilador gratuito AVR GCC.

En general todo el material necesario para el aprendizaje será basado en herramientas de código abierto, por ello se utilizará como sistema operativo Ubuntu 12.04, como IDE el Eclipse con el plugin AVR Eclipse Plugin disponible para descarga gratuita.

El tutorial presupone conocimientos previos de lenguaje C. No es necesario el conocimiento previo de programación de microcontroladores en general.

El desarrollo del material implementará un proceso de aprendizaje iterativo e incremental ocultando algunos detalles para facilitar la comprensión y se regresará, de a poco, sobre temas pasados para completarlo.

Cualquier consulta, duda o corrección será bienvenida en el foro de electrónica uControl (del amigo Ariel Palazzesi).

Índice General

Introducción

Herramientas necesarias

- Un microcontrolador ATmega8 o similar de 8 bits.
- Placa para pruebas o bien un protoboard.
- Compilador avr gcc y avr libc. Para windows está WinAVR y AVR Studio.
- Software del programador AVRDUDE
- Hardware del programador (ej.: Serial Prog, USBAsp, USBAsp Clone, etc)
- Para simulación y debug se puede utilizar en windows AVR Studio.

Mi primer programa

Comenzamos de 0, no? Bueno, aquí está el famoso “Hello World” en estos micros:

```
int main(void)
{
}
```

Un programa tan simple que es prácticamente imposible fallar :).

Todo programa de AVR tiene un punto de entrada llamado main y en el 99.9999% de los casos una vez que comienza a ejecutarse en el microcontrolador el código incluido en esta función es en un bucle (loop) infinito contenedor de todos los procesos implementados en el micro.
Vamos a completar un poco más el ejemplo:

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

int main(void)
{
      for (;;)
      {

      }
}
```

Este ejemplo sigue haciendo tantas cosas como el anterior, es decir nada, el micro inicia y entra en un loop infinito en la sentencia for.
El lector advirtió seguramente la diferencia entre ambos códigos en esta línea:

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

Esta directiva #include del pre-procesador de C incluye en forma selectiva el archivo encabezado (header) con las definiciones de registros de entrada/salida, vectores y bits relativos al microcontrolador elegido.
Estas definiciones nos ayudan a evitar plagar nuestro código con los valores numéricos y usar nombres más fáciles de leer y recordar.

En un Atmega8, por ej., el puerto D está definido de la siguiente manera en iom8.h

```
/* Port D */
#define PIND     _SFR_IO8(0x10)
#define DDRD     _SFR_IO8(0x11)
#define PORTD    _SFR_IO8(0x12)
```

PIND es el **P**ort **IN**put del puerto **D**.
DDRD es el **D**ata **D**irection **R**egister del puerto **D**.
PORTD es obviamente, el puerto (**PORT**) **D**.

DDRD sirve para especificar la dirección (sentido) de los datos de todo el puerto D, para especificar como **salida** un bit (pin del microcontrolador) en particular se podría escribir lo siguiente:

```
DDRD = 0x01;
```

Nota: Es conveniente conceptualizar a DDR como un registro que habilita -o no- a un pin de un puerto como salida.

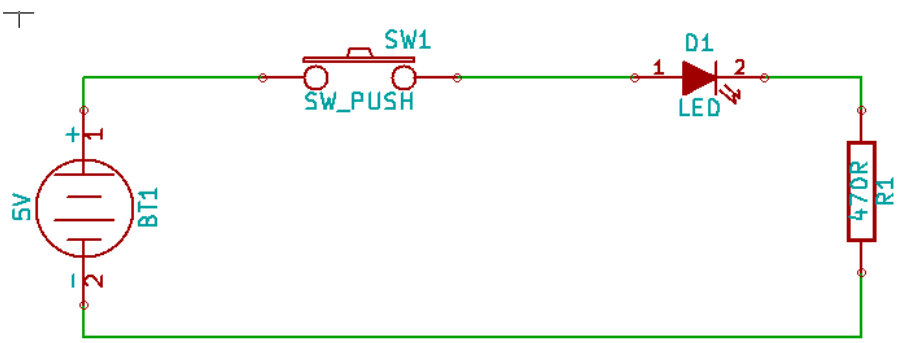
Cuando el pin de un puerto está configurado como salida, **PORTD** sirve para establecer su valor lógico (1=nivel alto, 0=nivel bajo).

Cuando el pin de un puerto está configurado como entrada, **PORTD** sirve para configurar la resistencia pull-up interna (de unos 50k aprox.). (1=habilitada, 0=deshabilitada). Si no se habilita el pull-up el pin queda en estado Tri-state (alta impedancia).

Si se establece el valor de **PUD** (pull-up disable) del registro **MCUCR** a 1, se deshabilitan las pull-up en todos los puertos y no se pueden habilitar con **DDRx** y **PORTx**.

PIND sirve para leer el valor lógico de un pin de un puerto que esté configurado como entrada.

Mi primer circuito



El objetivo de esta prueba será aprender lo necesario para reemplazar el pulsador SW1 por una acción automatizada en un microcontrolador.

La corriente suministrada por el micro en cada pin alcanza perfectamente para encender un led limitando la corriente con una resistencia, el valor de dicha resistencia está dado por la fórmula:

$R = \frac{(Vin - VLed)}{I}$	V_{in}: Voltaje de alimentación del micro, en nuestro caso, utilizaremos una alimentación regulada de 5v. V_{led}: Voltaje de led, normalmente 2v, 4v para los leds azules. I: Corriente a suministrar al led, 10mA o 20mA según la luminosidad deseada y tensión de alimentación.

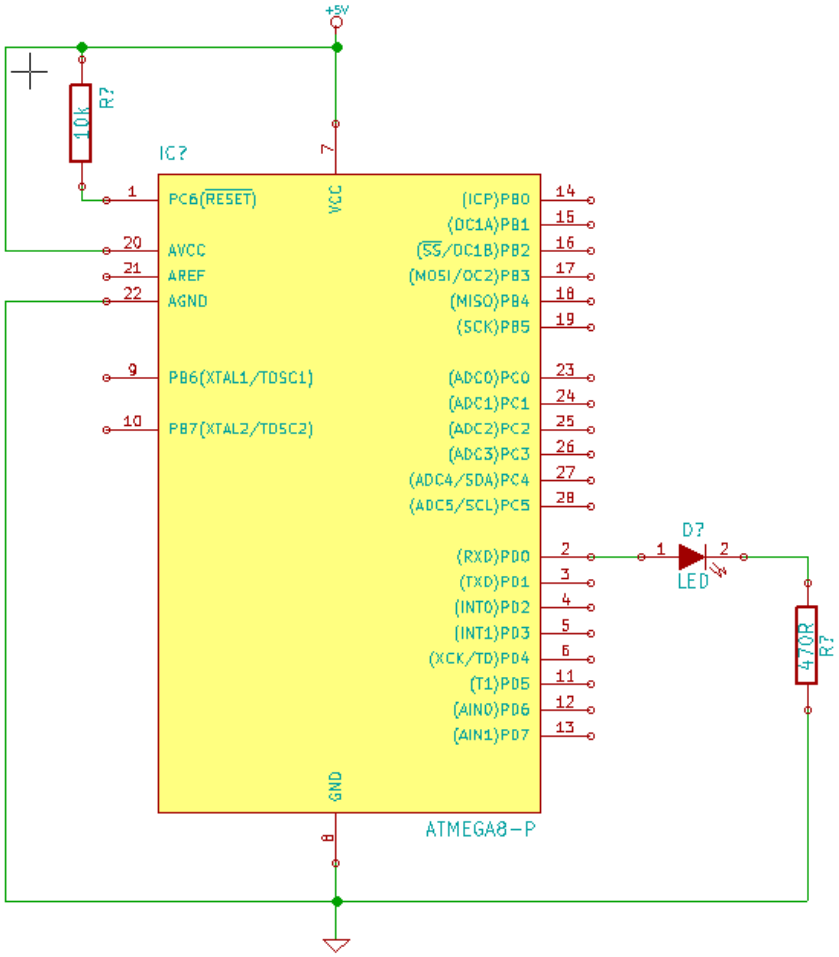
Para saber los límites de corriente por puerto de nuestro micro elegido revisar el datasheet (Electrical Characteristics), para un ATmega8 el valor pico máximo es 40mA por pin, pero:

1. La suma de corriente suministrada en todos los puertos no debe superar los 300mA.
2. La suma de corriente suministrada en los puertos C0-C5, no debe superar los 100mA.
3. La suma de corriente suministrada en los puertos B0-B7, C6, D0-D7 y XTAL2, no debe superar los 200mA.

Nota: Es imprescindible **leer** y tener a mano el **datasheet** (hoja de datos) del componente. Los valores máximos en el datasheet **no** son valores recomendables, siempre que se pueda se debe usar un valor inferior para operar en una zona con un margen de seguridad que redundará en el largo plazo en mayor duración y confiabilidad del componente.

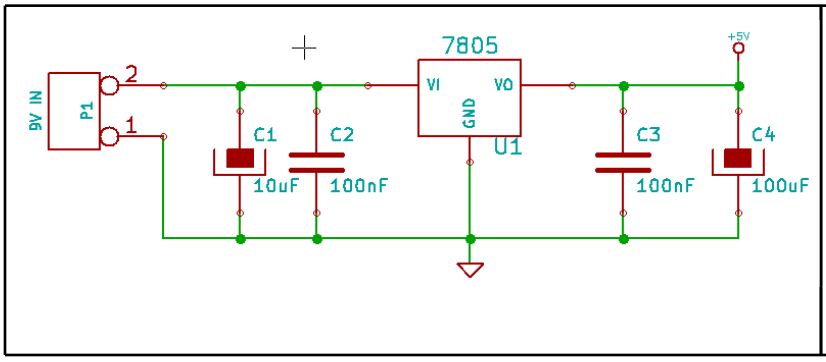
Para todos nuestros ejemplos con Vcc=5V el valor máximo que usaremos será 20mA.

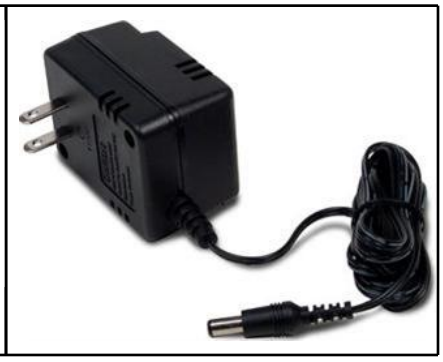
Reemplazando SW1 por un micro el circuito quedaría así:



PC6 es el **RESET-pin**, un pulso de nivel bajo de al menos 1.5us genera un reset externo en el microcontrolador, en nuestro caso vamos a dejarlo a nivel alto por medio de una resistencia de 10k.

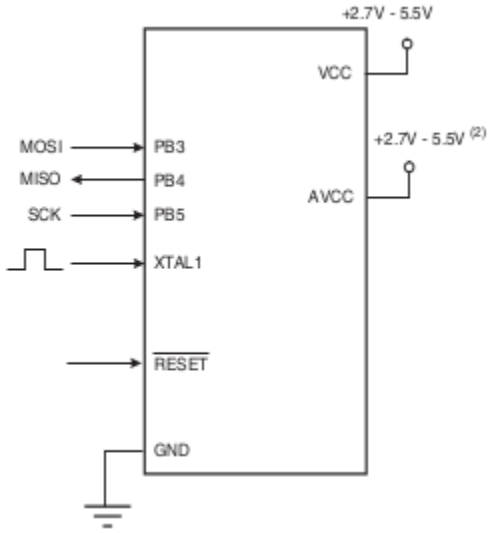
Para obtener los 5v en VCC utilizaremos una fuente común con transformador de 220VAC/ 110VAC a 9VCC 500mA, y un regulador de tensión como el 7805.





Programación del firmware

Para la grabación de nuestro programa utilizaremos la programación serial, según el datasheet del **ATmega8** estos son los puertos que intervienen:

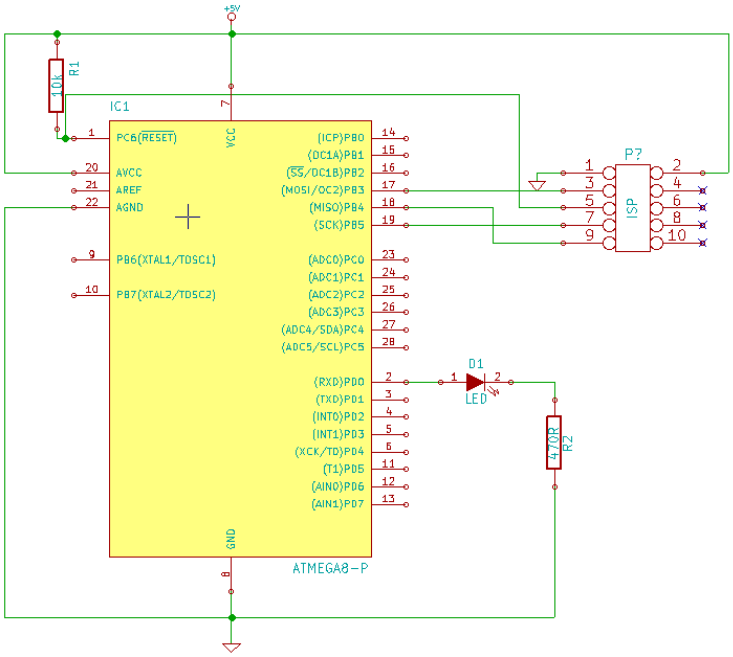


MOSI, **MISO**, **SCK** y **RESET** forman la interfaz de programación serial (**SPI**).

AVCC no debe quedar sin conectar al programar, al igual que **AGND** el momento de la ejecución.

Por ahora, al micro no se le conectará un cristal. Usaremos el oscilador interno como viene configurado en sus fuses al comprarlo. Este oscialdor RC en el caso de un ATmega8 es de **1 Mhz** (o 1.000.000 Hz)no es preciso pero servirá para aprender.

El circuito ejemplo entonces quedaría así:



Nota: Los cables usados para la programación SPI deben ser lo más cortos posibles.

Led ON

Continuando con el objetivo propuesto de reemplazar el pulsador por un micro, vamos a escribir el código que encenderá nuestro led conectado a PD0.

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

int main(void)
{
	DDRD = 0x01;  // pone al pin 0 del puerto D como salida el
resto como entradas.
	PORTD = 0x01; // pone en nivel alto al pin 0 del puerto D

	for (;;)
	{

	}
}
```

Nota: La segunda instrucción PORTD = 0x01 activa las resistencias **pull-ups** de **PD1** a **PD7**, porque están definidas como entradas.

Creamos el proyecto C llamado **LedOn** en Eclipse (con el AVR Eclipse Plugin) como un **AVR Cross Target Application** usando el **AVR-GCC Toolchain**.

La instalación del toolchain en Ubuntu 12.04 se logra ejecutando lo siguiente:

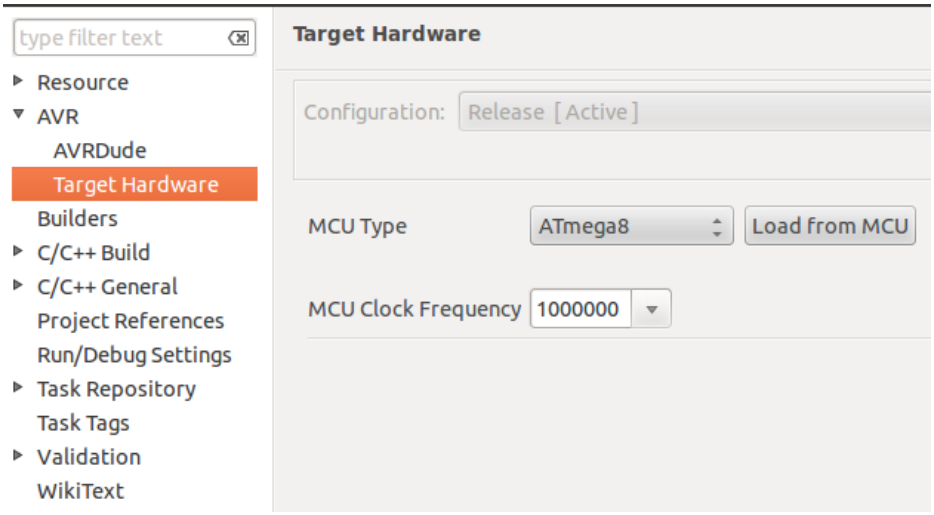
```
$ sudo apt-get install gcc-avr avr-libc avrdude
```

Una vez instalado para usar el programador USBasp será necesario darle permisos a avrdude para que pueda acceder al puerto USB, hay varias formas, esta es una:

```
$ sudo chmod a+s /usr/bin/avrdude
```

Nota: Las instrucciones de instalación de **Eclipse** y **AVR Eclipse Plugin** están en sus respectivas guías de usuario.

Para grabar el programa el proyecto debe tener configurado el micro destino en nuestro caso como ya se dijo es un **ATmega8** y **1 Mhz** de oscilador interno.



Al compilar en Release el proyecto obtendremos esta salida en la Console:

```
**** Build of configuration Release for project tutorial01 ****

make all
Building file: ../main.c
Invoking: AVR Compiler
avr-gcc -Wall -Os -fpack-struct -fshort-enums -std=gnu99 -funsigned-char -funsigned-bitfields -mmcu=atmega8 -
DF_CPU=1000000UL -MMD -MP -MF"main.d" -MT"main.d" -c -o "main.o" "../main.c"
Finished building: ../main.c

Building target: tutorial01.elf
Invoking: AVR C Linker
avr-gcc -Wl,-Map,tutorial01.map -mmcu=atmega8 -o "tutorial01.elf"  ./main.o
Finished building target: tutorial01.elf

Invoking: AVR Create Extended Listing
avr-objdump -h -S tutorial01.elf  >"tutorial01.lss"
Finished building: tutorial01.lss

Create Flash image (ihex format)
avr-objcopy -R .eeprom -O ihex tutorial01.elf  "tutorial01.hex"
Finished building: tutorial01.hex

Create eeprom image (ihex format)
avr-objcopy -j .eeprom --no-change-warnings --change-section-lma .eeprom=0 -O ihex tutorial01.elf  "tutorial01.eep"
Finished building: tutorial01.eep

Invoking: Print Size
avr-size --format=avr --mcu=atmega8 tutorial01.elf
AVR Memory Usage
----------------
Device: atmega8

Program:      84 bytes (1.0% Full)
(.text + .data + .bootloader)

Data:         0 bytes (0.0% Full)
(.data + .bss + .noinit)
```

```
Finished building: sizedummy

**** Build Finished ****
```

Como se puede ver el IDE nos preparó el archivo HEX utilizando el toolchain con los parámetros predefinidos a nivel proyecto.

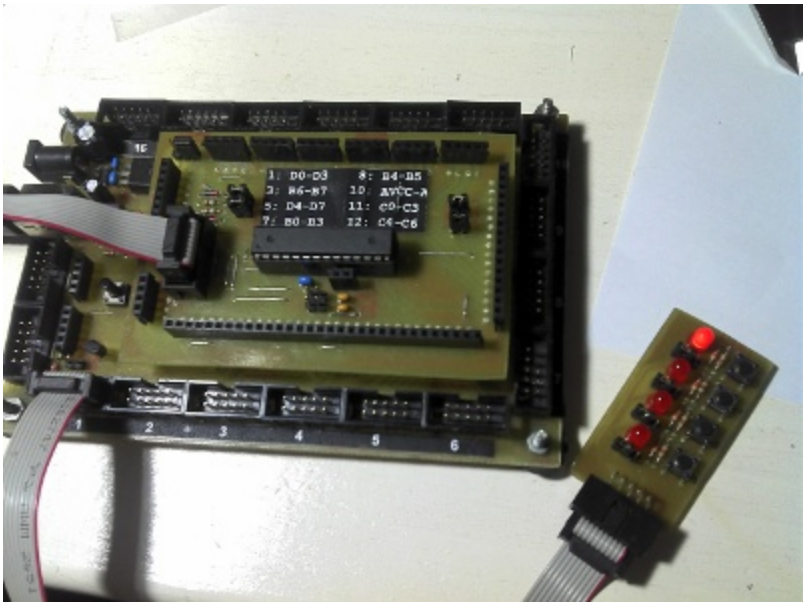
El comando más importante ejecutado aquí es la compilación:

```
avr-gcc -Wall -Os -fpack-struct -fshort-enums -std=gnu99 -funsigned-char -funsigned-bitfields -mmcu=atmega8 -DF_CPU=1000000UL -MMD -MP -MF"main.d" -MT"main.d" -c -o "main.o" "../main.c"
```

El parámetro **-mmcu=atmega8** y el define **-DF_CPU** lo obtiene de la definición previa (config. AVR/Taget Hardware).

Seleccionamos el proyecto y hacemos click en *Upload current project to Atmel target MCU*.

Si todo fue bien hasta aquí, al terminar de grabar deberíamos tener un led encendido y el micro en un loop eterno.-



Blinking LED

Este proyecto es una prueba básica de programación de microcontroladores.
Su función? Encender y apagar un led.

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

int main(void)
{
      DDRD = 0x01;  // DD0 como salida, el resto entradas.

      for (;;)
      {
            PORTD = 0x01; // PD0=1, el resto a 0.
            _delay_ms(100);
            PORTD = 0x00; // PD0=0, el resto a 0.
            _delay_ms(100);
      }
}
```

La función **_delay_ms** de <util/delay.h> posee los cálculos retardo en ciclos de instrucción necesarios para lograr la espera exacta de 100ms.

Notar que estamos afectando con nuestras instrucciones a todos los bits del puerto D, para trabajar solo con DD0 se usa el desplazamiento de bits y los operadores & (AND) y | (OR), ejemplo:

```
DDRD |= (1<<DD0); // DD0=1 (salida), el resto no cambia
```

Donde: 1<<DD0 = 1<<0 = 0b00000001

De forma expandida la expresión anterior quedaría entonces:

```
DDRD = DDRD | 0b00000001;
```

Quiere decir que el nuevo valor de DDRD será igual a el OR entre el valor anterior de DDRD y 0b00000001

Analogamente podemos hacer esto con PORTD = 0x01 utilizando esta expresión:

```
PORTD |= (1<<PD0); // PD0=1, el resto no cambia.
```

En el otro caso (para PD0=0) se utiliza una mascara (AND) para ello se obtiene el complemento a dos por medio del NOT (operador ~)

```
PORTD &= ~(1<<PD0); // PD0=0, el resto no cambia.
```

Donde: ~(1<<PD0) = ~(1<<0) = 0b11111110

De forma expandida la expresión anterior quedaría entonces:

```
PORTD = PORTD & 0b11111110;
```

Quiere decir que el nuevo valor de PORTD será igual a el AND entre el valor anterior de PORTD y 0b11111110

Y dado que sabemos que...

& (AND)	| (OR)	^ (XOR)
0 & 0 = 0	0 | 0 = 0	0 ^ 0 = 0
0 & 1 = 0	0 | 1 = 1	0 ^ 1 = 1
1 & 0 = 0	1 | 0 = 1	1 ^ 0 = 1
1 & 1 = 1	1 | 1 = 1	1 ^ 1 = 0

simplificaremos nuestro código usando XOR para alternar el valor entre 1 y 0 por cada iteración del bucle infinito y podemos quitar la segunda instrucción _delay_ms.

Este sería nuestro código del blinking led simplificado:

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

int main(void)
{
      DDRD |= (1<<DD0); // DD0=1 (salida), el resto no cambia

      for (;;)
      {
            PORTD ^= (1<<PD0); // alterna valor de PD0
            _delay_ms(100);
      }
}
```

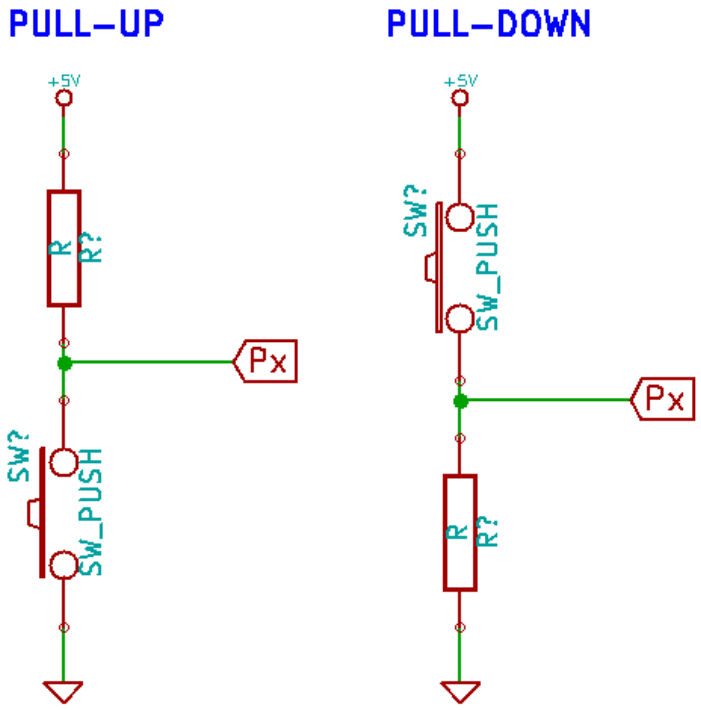
Nota: La función **_delay_ms** de **delay.h** tiene una la limitante de retardo y está relacionada con la velocidad del oscilador, $max_{ms} = 262.14 / F_{CPU}(en\ Mhz)$. En este caso, dado que F_CPU en este ejemplo es 1 Mhz el valor máximo de parámetro es 262.14ms/1 = 262ms.

Uso de entradas

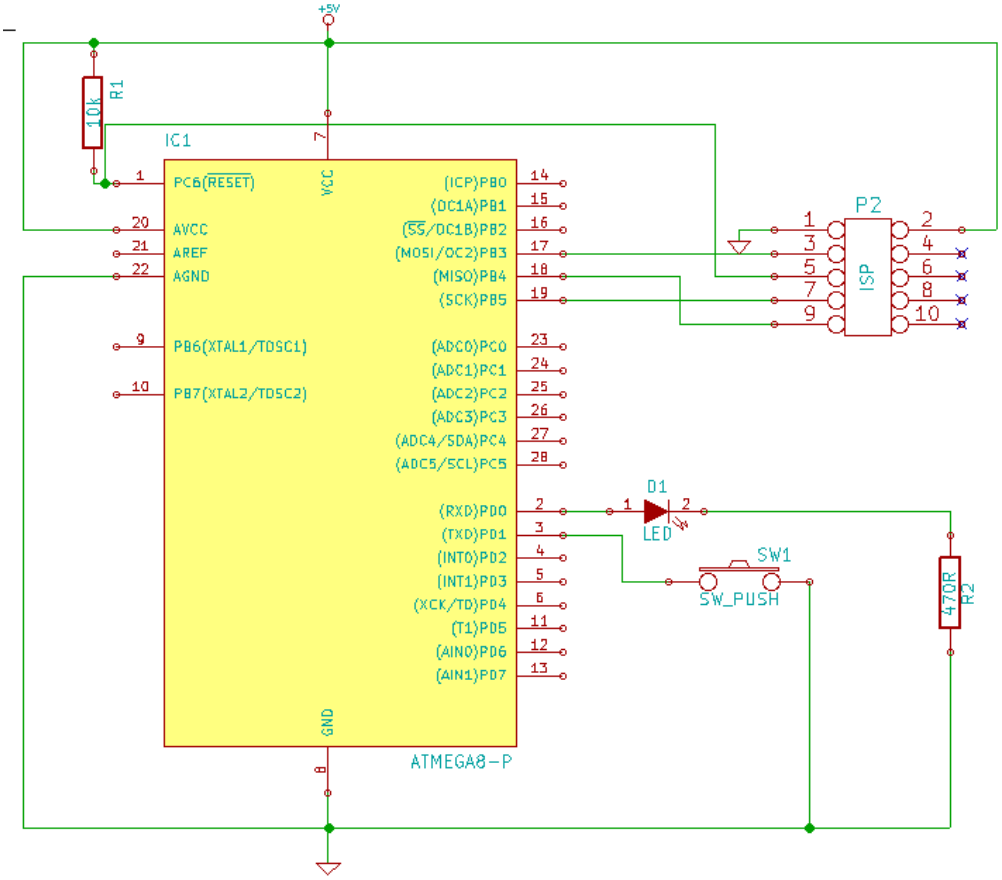
Cuando el microcontrolador necesita enterarse de eventos externos como por ejemplo si el usuario presionó sobre un pulsador se debe configurar un pin como entrada. Esto se logra como ya hemos visto con el registro de dirección de datos (DDR) colocando un 1 lógico en el bit del puerto al cual está conectado el dispositivo de entrada.

Como dato adicional, si únicamente activamos el pin de un puerto como entrada esto dejaría en Tri-State, en ese caso cualquier ruido externo podría darnos 1 o 0 aleatorios ya que el pin estaría “flotando”, para evitar este comportamiento no deseado se debe configurar un pull-up o pull-down, y como vimos antes el microcontrolador cuenta con pull-up internas que para este ejemplo sirven perfectamente.

Estas son las posibles configuraciones de entradas digitales:



Retomando entonces el ejemplo Led ON con el led conectado en PD0 agregaremos un pulsador conectado al pin PD1.



Actualizando el código de Led ON entonces quedaría:

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

int main(void)
{
	DDRD |= (1<<DDD0);	// DDD0=1=salida
	DDRD &= ~(1<<DDD1);	// DDD1=0=entrada
	PORTD |= (1<<PD1);	// pull-up activa para PD1

	PORTD &= ~(1<<PD0);	// inicia con el led apagado
	for (;;)
	{
		if (!(PIND & (1<<PD1)))	//pulsador?
			PORTD |= (1<<PD0);	//enciende el led
		else
			PORTD &= ~(1<<PD0);	//apaga el led
```

```
        }
}
```

Al grabar este programa el led en PD0 permanecerá apagado mientras que al presionar el pulsador conectado a PD1 lo enciende.

Analizamos la línea de la lectura del pin 1 del puerto D:

```
            if (!(PIND & (1<<PD1)))     //pulsador?
```

Debido a que habilitamos la resistencia pull-up interna sin tener presionado el pulsador PIND & (1<<PD1) se leerá un 1 lógico.

Si a esta altura el lector intuye que la sintaxis es un tanto críptica, pues, le doy la razón, aunque el maravilloso C tiene los #define que nos ayudan a poder leer el código de forma más... humana?

Usando las funciones **bit_is_clear**, **bit_is_set** definidas en **sfr_defs.h.**

```
#define _BV(bit) (1 << (bit))

#define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))

#define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
```

La expresión condicional de más arriba se podría reescribir así:

```
            if (bit_is_clear(PIND, PD1))     //pulsador?
```

Y se podría usar estos defines para las otras expresiones, le dejo el placer de ser creativo en esa tarea :)

Rebotes

Los pulsadores son interruptores mecánicos imperfectos, en el cierre de contactos se producen micro-rebotes y la señal que recibe el microcontrolador oscila entre 0v y 5v por un intervalo de 20 a 40 milisegundos aproximadamente.

Este comportamiento provoca en nuestros programas una pulsación multiple no deseada (la mayoría de las veces).

Existen diferentes técnicas para parchar este problema, veremos la más sencilla y es la de preguntar si el botón fue pulsado, hacer una espera de 20ms y luego preguntar si el botón sigue pulsado.

Consideremos este ejemplo

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

int main(void)
{
	DDRD |= (1<<DD0);	// DD0=1=salida
	DDRD &= ~(1<<DD1);	// DD1=0=entrada
	PORTD |= (1<<PD1);	// pull-up activa para PD1

	PORTD &= ~(1<<PD0);	// inicia con el led apagado
	for (;;)
	{
		if (bit_is_clear(PIND, PD1))	//pulsador?
		{
			PORTD ^= (1<<PD0);		//alterna el led
		}
	}
}
```

Al iniciar este programa el Led estará apagado, al pulsar el botón conectado en PD1 lo encenderá y apagará de forma alternada, pero no siempre será así, se podrá observar que a veces el Led cambia al estado siguiente pero otras veces no.

Para solucionar entonces el problema cambiamos por esto:

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

int main(void)
{
      DDRD |= (1<<DDD0);    // DD0=1=salida
      DDRD &= ~(1<<DDD1);   // DD1=0=entrada
      PORTD |= (1<<PD1);    // pull-up activa para PD1

      PORTD &= ~(1<<PD0);   // inicia con el led apagado
      for (;;)
      {
            if (bit_is_clear(PIND, PD1))     //pulsado?
            {
                  _delay_ms(20);
                  if (bit_is_clear(PIND, PD1))     //pulsado?
                  {
                        PORTD ^= (1<<PD0);    //alterna el led
                        while(bit_is_clear(PIND, PD1)); //espera a que
el usuario deje de pulsar.
                  }
            }
      }
}
```

Hay otros métodos, tan fáciles como colocar un condensador entre los contactos del botón, y tan complejos como usar un timer (tema que veremos más adelante).

Memorias

Flash

Nuestro ATmega8 tiene distintos tipos de memorias y una de ellas es la Flash EEPROM (8Kbytes) para almacenamiento del programa (soporta aprox. 10.000 ciclos de escritura/ borrado eléctrico). Las instrucciones de AVR (assembler) utilizan 16/32 bits, la Flash está organizada en 4k x 16 bits por lo que el PC (program counter) puede direccionar 4K ($000 a $FFF) con sus 12 bits de ancho. Al final de esta memoria se encuentra una sección para el bootloader.

EEPROM

Contiene 512 bytes de datos en una memoria EEPROM separada de la Flash y organizada en simples bytes que se pueden leer o escribir con una duración estimada en 100.000 ciclos de lectura/escritura.

SRAM

Memoria volatil (su contenido se pierde al quitar al fuente de alimentación) de 1Kbyte compuesta de:

- Registros de propósito general del CPU (R0 a R31), direcciones ($0000 a $001F)
- Registros de entrada y salida ($00 a $3F), direcciones ($0020 a $005F)
- SRAM interna, direcciones ($0060 a $045F)

En esta memoria se almacenan las variables y la Pila de programa (o Stack).

Debido a la poca memoria disponible es necesario utilizar todas las técnicas necesarias para minimizar su utilización. El compilador ayudará en gran medida en esta tarea reasignando una variable a un registro del CPU, almacenando las constantes en la Flash, eliminando variables no utilizadas, etc. Sin embargo se debe tener especial cuidado con este tema y es aconsejable usar siempre la opción de compilador -Os que optimiza el código en tamaño y diseñar nuestro programa de manera consistente con estas limitantes.
Los AVR permiten variables almacenadas en SRAM, FLASH (constantes) y EEPROM usando modificadores de declaración y funciones para lectura/escritura. Además es posible indicarle al compilador que no debe optimizar una variable obligando a leer y escribir de la memoria. Además se realizan optimizaciones de acuerdo al ambito (local/global).

Variables

Las variables son almacenadas en la SRAM como un conjunto de x bytes se utilizan dentro de nuestro código C con una representación de acuerdo al tipo de dato que se desea manejar.
Los tipos de datos en AVR son:

Tipo	Tamaño (bits)	Rango
char	8	-128 a 127
int8_t	8	-128 a 127
uint8_t	8	0 a 255
int16_t	16	-32768 a 32767
uint16_t	16	0 a 65535
int32_t	32	-2147483648 a 2147483647
uint32_t	32	0 a 4294967295
int64_t	64	-(2^64)/2 a (2^64)/2+1
uint64_t	64	0 a 2^64
float	32	+-1.175e-38 a +-3.402e38
double	32	+-1.175e-38 a +-3.402e38

Si bien existen los tipos de C como int, short, etc, no se recomienda su uso. Como se sabe un String o cadena de caracteres no existen en este lenguaje pero se dispone de los vectores de char (o int8_t) para almacenarlas.

Campos de bits (Bit Fields)

Cuando se necesita guardar datos de un tamaño menor al byte es posible agrupar bits utilizando un struct

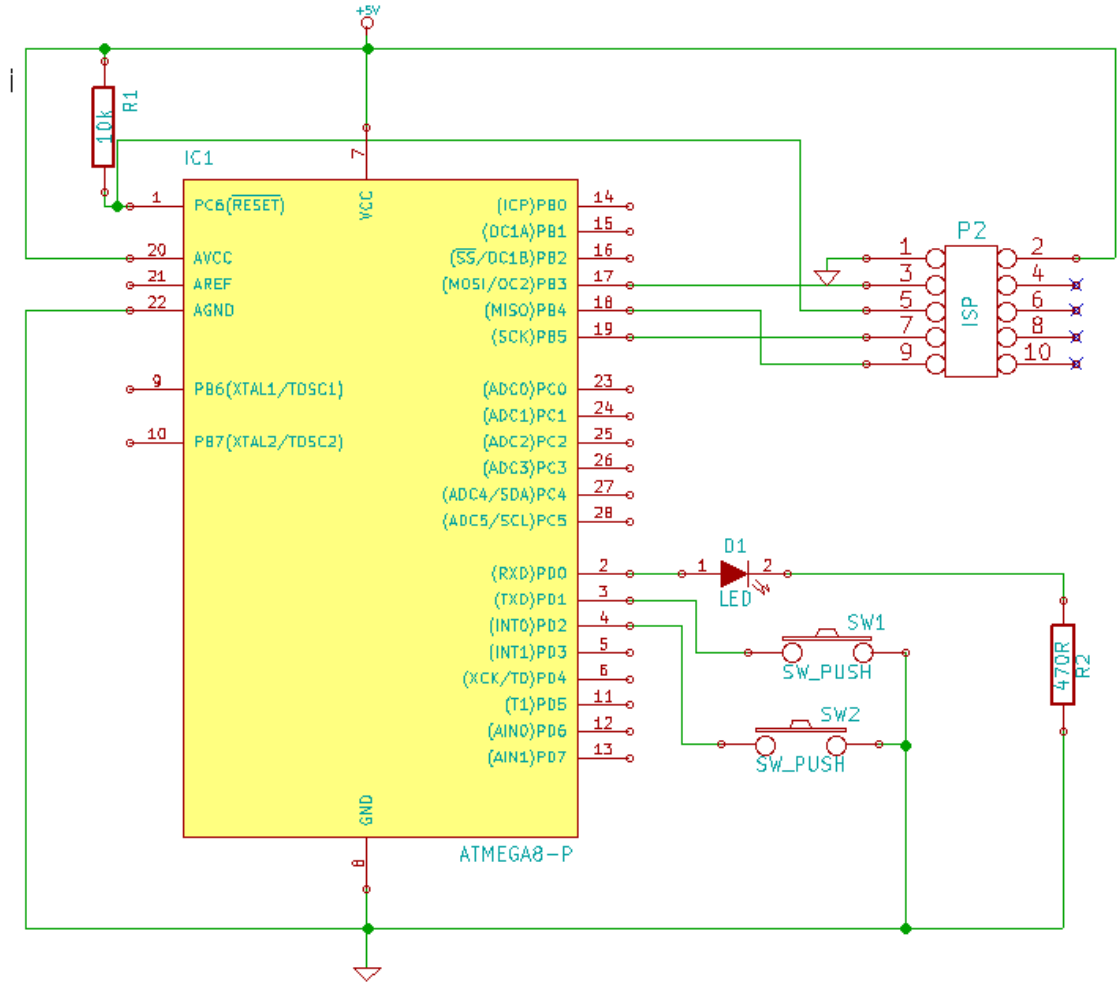
```
struct
{
        unsigned estado1 : 1;
        unsigned estado2 : 1;
        unsigned dosbits : 2;
} v;
```

Y luego es posible acceder a cada bit de forma separada:

```
        v.estado1 = 1;
        v.estado2 = 0;
        v.dosbits = 0x3;
```

Uso práctico

Utilizaremos una variable del tipo uint8_t para almacenar la cantidad de destellos del led en PD0 cuando se pulse SW1. Para ello agregaremos otro pulsador SW2 conectado a PD2 que incrementará la variable “cantidad de parpadeos”.



Código:

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

int main(void)
{
        DDRD |= (1<<DDD0);    // PD0=1=salida
        DDRD &= ~(1<<DDD1); // PD1=0=entrada
        DDRD &= ~(1<<DDD2); // PD2=0=entrada
```

```
	PORTD |= (1<<PD1);    // pull-up activa para PD1
	PORTD |= (1<<PD2);    // pull-up activa para PD2
	uint8_t tiempo = 0;
	PORTD &= ~(1<<PD0); // inicia con el led apagado
	for (;;)
	{
		if (bit_is_clear(PIND, PD1))
		{
			_delay_ms(20);
			if (bit_is_clear(PIND, PD1))
			{
				for (uint8_t i = 0; i < tiempo; i++)
				{
					PORTD ^= (1 << PD0); //alterna el led
					_delay_ms(20);
				}
				PORTD &= ~(1 << PD0); // apaga el led
				loop_until_bit_is_set(PIND, PD1);
			}
		}
		if (bit_is_clear(PIND, PD2))
		{
			_delay_ms(20);
			if (bit_is_clear(PIND, PD2))
			{
				tiempo += 5;
				loop_until_bit_is_set(PIND, PD2);
			}
		}
	}
}
```

Al pulsar sobre el interruptor de PD1 el led de PD0 destellará con un intervalo de 20ms t veces.

Nota: La llamada a **loop_until_bit_is_set** de **sfr_defs.h** tiene la misma función que realizamos en nuestro ejemplo anterior.

Vamos a hacer un poco más legible el código, para ello escribiremos estos #defines:

```
#define clear_bit(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define set_bit(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define toogle_bit(sfr, bit) (_SFR_BYTE(sfr) ^= _BV(bit))
```

El código anterior quedaría así:

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

#define clear_bit(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define set_bit(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
```

```
#define toogle_bit(sfr, bit) (_SFR_BYTE(sfr) ^= _BV(bit))

int main(void)
{
	set_bit(DDRD, DDD0);  // PD0=1=salida
	clear_bit(DDRD, DDD1);		// PD1=0=entrada
	clear_bit(DDRD, DDD2);		// PD2=0=entrada
	set_bit(PORTD, PD1);  // pull-up activa para PD1
	set_bit(PORTD, PD2);  // pull-up activa para PD2

	uint8_t tiempo = 0;

	clear_bit(PORTD, PD0);		// inicia con el led apagado
	for (;;)
	{
		if (bit_is_clear(PIND, PD1))
		{
			_delay_ms(20);
			if (bit_is_clear(PIND, PD1))
			{
				for (uint8_t i = 0; i < tiempo; i++)
				{
					toogle_bit(PORTD, PD0); //alterna el led
					_delay_ms(20);
				}
				clear_bit(PORTD, PD0); // apaga el led
				loop_until_bit_is_set(PIND, PD1);
			}
		}
		if (bit_is_clear(PIND, PD2))
		{
			_delay_ms(20);
			if (bit_is_clear(PIND, PD2))
			{
				tiempo += 5;
				loop_until_bit_is_set(PIND, PD2);
			}
		}
	}
}
```

USART

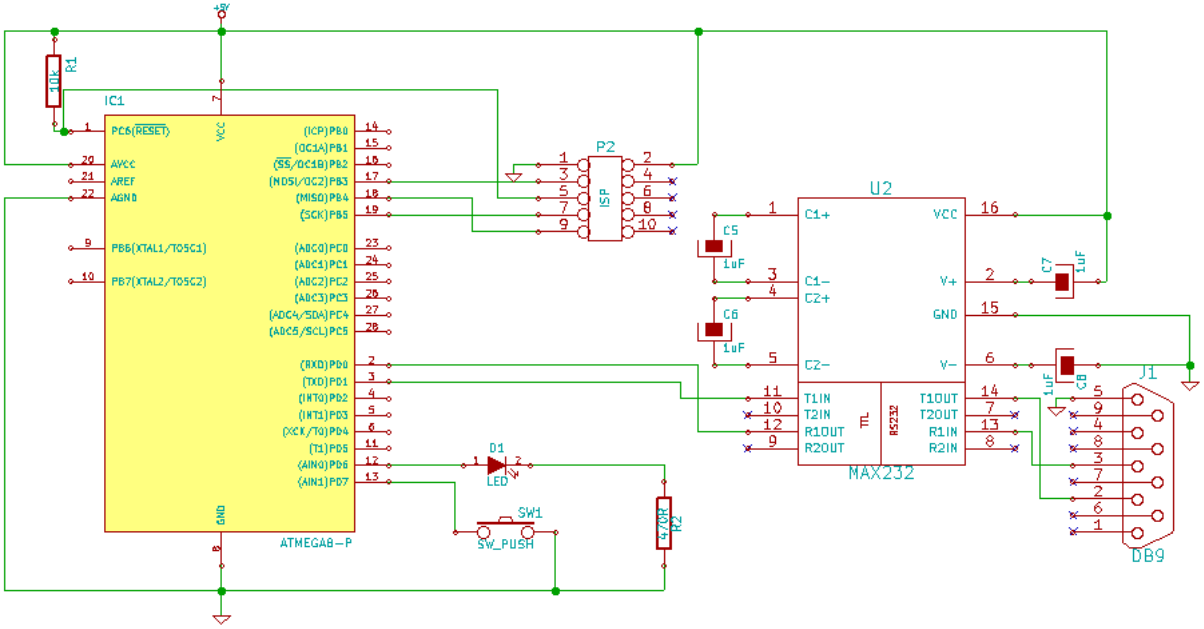
Los gran mayoría de las familias de AVR poseen al menos una interfaz USART, la cual sirve para enviar y recibir datos seriales desde el microcontrolador a un dispositivo (una PC, por ejemplo) o a otro AVR.
Esta interfaz usa solo tres pines (RXD, TXD y GND) enviando/recibiendo los datos usando una frecuencia pre-acordada y nos servirá en el resto del tutorial para hacer el debug de nuestros programas a modo de log a nuestras PC usando un programa de emulación de terminal.

La especificación RS232

USART es normalmente relacionada con la especificación RS232, mientras que la comunicación USART establece niveles lógicos de 3 a 5V, la comunicación RS232 usa +3V a +25V para un **0(ESPACIO)** digital y -3V a -25V para un **1(MARCA)** digital.

USART presentes en dos AVRs pueden interconectarse conectando RXD y TXD de forma directa; si los dispositivos están usando RS232 se debe usar un circuito conversor de niveles. En este caso vamos a conectar el ATmega8 a la PC para ello será necesario usar el MAX232 (o el MAX3232 si se trabaja con VDD=3V)

Las notebook dejaron de venderse practicamente con puerto serial, es por ello que será necesario para estos casos usar un conversor USB-Serie bidireccional. Nuestro circuito de prueba sería el siguiente:



Inicializando la USART

Para la inicialización se utilizan los registros UCSRB para configurar el modo de funcionamiento de la interfaz (envío, recepción o ambas); UCSRC para establecer si es sincrónica o asincrónica, cantidad de bits del dato, la paridad, bit(s) de stop, etc.

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

int main(void)
{
	UCSRB |= (1 << RXEN) | (1 << TXEN);// Inicializa la USART para envio y recepción.

	UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Asincrónica, 8,N,1

	for (;;)  { }
}
```

Lo que está faltando es indicarle la velocidad y eso está dado por el baudrate que fijaremos para este ejemplo en 4800 baudios, pero el baudrate no se carga con ese valor en el registro de UBRRH y UBRRL sino que se lo calcula con esta fórmula:

$$ValorUBRR = \frac{Fcpu}{BAUDRATE*16} - 1$$

Dado que estamos trabajando con un oscialdor interno a 1 Mhz y BAUDRATE será 4800:

$$ValorUBRR = \frac{1.000.000}{4800*16} - 1 = 12$$

Colocando estos cálculos en #defines:

```
#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)
```

Y establecemos el valor de UBRR así:

```
UBRRH = (BAUD_PRESCALE >> 8);
UBRRL = BAUD_PRESCALE;
```

Nota: La velocidad de 4800 baudios es la máxima para 1Mhz si se desea tener un error por debajo del 0.2%. Consultar el datasheet los diferentes valores de UBRR y ratios de errores asociados.

El código de la inicialización quedaría así:

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

#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)

int main(void)
{
        UCSRB |= (1 << RXEN) | (1 << TXEN);// Inicializa la USART para envio y recepción.

        UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Asincrónica, 8,N,1

        UBRRH = (BAUD_PRESCALE >> 8);
        UBRRL = BAUD_PRESCALE;

        for (;;)
        {
        }
}
```

Y ya estamos listos para enviar y recibir datos!

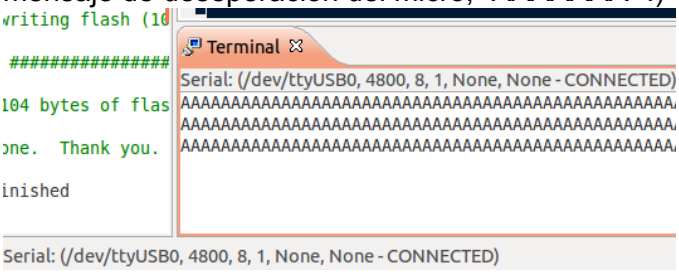
Envío de datos

Nuestro querido AVR va comunicarse con el mundo exterior, para ello tenemos que escribir en el registro **UDR** un carácter ASCII (para que sea posible leerlo en pantalla), pero antes de escribir en UDR deberíamos saber que no hay un dato anterior enviandose, para ello está el registro de estado y control **UCSRA**, bit **UDRE** el cual se pone en 1 cuando el buffer de transmición está vacío (es decir, podemos transmitir).

Dentro del loop infinito agregamos:

```
while (( UCSRA & (1 << UDRE ) ) == 0) {};
UDR = 65;
_delay_ms(50);
```

Al grabar el programa y conectar a la pc el cable db9 o el conversor usb-serie abrimos el programa terminal que tengamos a mano con los parámetros 4800, 8, N, 1 y veremos el mensaje de deseperación del micro, “AAAAAAA” :)



Recepción de datos

Para recibir los datos transmitidos desde la PC por el pin RXD, vamos a leer el contenido de **UDR**, pero antes haremos un bucle de espera consultando al UCSRA, bit RXC el cual se pone en 1 cuando el buffer de recepción está lleno:

```
while (( UCSRA & (1 << RXC ) ) == 0) {};
uint8_t datoLeido = UDR;
```

Ejemplo “ECO eco eco...”

Para unir todos los conceptos de USART implementaremos un típico Echo o consola eco de caracteres donde la tecla que se pulsa en la terminal el microcontrolador la recibe y responde con el mismo carácter.

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

#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)

int main(void)
{
	UCSRB |= (1 << RXEN) | (1 << TXEN);  // Inicializa la USART para envio y recepción.

	UCSRC |=  (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Asincrónica, 8,N,1

	UBRRH = (BAUD_PRESCALE >> 8);
	UBRRL = BAUD_PRESCALE;

	for (;;)
	{
		while (( UCSRA & (1 << RXC ) ) == 0) {};
		uint8_t datoLeido = UDR;

		while (( UCSRA & (1 << UDRE ) ) == 0) {};
		UDR = datoLeido;
	}
}
```

Ejemplo control remoto

Usando el circuito propuesto aquí encenderemos el LED de PD6 al pulsar 0 en la terminal y cuando se pulse el interruptor en PD7 el microcontrolador enviará un “Hola mundo AVR”.

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

#define clear_bit(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define set_bit(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define toogle_bit(sfr, bit) (_SFR_BYTE(sfr) ^= _BV(bit))

#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)

int main(void)
{
	clear_bit(DDRD, DDD7);
	set_bit(PORTD, PD7);   //PD7 (pulsador)

	set_bit(DDRD, DDD6);    //PD6 (led)
	clear_bit(PORTD, PD6); //led apagado

	UCSRB |= (1 << RXEN) | (1 << TXEN);              // Inicializa la USART para envio y recepción.
	UCSRC |=  (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Asincrónica, 8,N,1
	UBRRH = (BAUD_PRESCALE >> 8);
	UBRRL = BAUD_PRESCALE;

	char mensaje[] = "Hola mundo AVR\r\n";

	for (;;)
	{
		if (bit_is_clear(PIND, PD7))
		{
			_delay_ms(20);
			if (bit_is_clear(PIND, PD7))
			{
				char *s = mensaje;
				while (*s != 0)
				{
					while (( UCSRA & (1 << UDRE ) ) == 0) {};
						UDR = *s++;
				}
				loop_until_bit_is_set(PIND, PD7);
			}
		}
		if ( UCSRA & (1 << RXC ))
		{
			if (UDR == '0')
				toogle_bit(PORTD, PD6);
		}
	}
}
```

Diseño modular

Nuestro programa de comunicaciones funciona bien, hace su tarea digamos, pero si tuvieramos más comunicaciones en nuestro programa las líneas correspondientes al uso de los registros de USART estarían repetidas por todos lados, esta situación causa duplicación de código que dificulta el mantenimiento y hace más grande el ejecutable final.
Para tratar de solucionar estos temas en C se aplica el diseño modular y se obtienen los siguientes beneficios:

- Facilita el diseño descendente
- Disminuye la complejidad del algoritmo
- Disminuye el tamaño total del programa
- Reusabilidad: ahorro de tiempo de programación
- División de la programación entre un equipo de programadores (menor tiempo de desarrollo)
- Facilidad en la depuración: comprobación individual de los módulos
- Programas más fáciles de modificar
- Estructuración en bibliotecas específicas (biblioteca de módulos)

Aunque no es la solución ideal, debido al acoplamiento por argumento de funciones, es mejor que nada.

Llevamos entonces las funciones de USART a un archivo usart.c separado del main.c

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

void usart_init(uint16_t bautrate)
{
	UCSRB |= (1 << RXEN) | (1 << TXEN);   // Inicializa la USART para envio y recepción.
	UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Asincrónica, 8,N,1
	UBRRH = (bautrate >> 8);
	UBRRL = bautrate;
}

void usart_putc(char c)
{
	while (!(UCSRA & (1 << UDRE))) /* espera a que UDR esté listo... */
	{
	}

	UDR = c;
}
```

```
void usart_puts(char *s)
{
	while (*s)
		usart_putc(*s++);
}

char usart_getc()
{
	while (!(UCSRA & (1 << RXC)))		/* espera a tener el dato en UDR */
	{

	}
	return UDR;
}
```

Y hacemos su archivo header usart.h:

```
#ifndef _USART_H_
#define _USART_H_

#define usart_kbhit() (UCSRA & (1 << RXC))
void usart_init(uint16_t );
void usart_putc(char );
void usart_puts(char *);
char usart_getc();

#endif
```

Nuestro main.c quedaría así:

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

#include "usart.h"

#define clear_bit(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define set_bit(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define toogle_bit(sfr, bit) (_SFR_BYTE(sfr) ^= _BV(bit))

#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)

int main(void)
{
	clear_bit(DDRD, DDD7);
	set_bit(PORTD, PD7);	//PD7 (pulsador)

	set_bit(DDRD, DDD6);	//PD6 (led)
	clear_bit(PORTD, PD6); //led apagado

	usart_init(BAUD_PRESCALE);

	char mensaje[] = "Hola mundo AVR\r\n";

	for (;;)
	{
		if (bit_is_clear(PIND, PD7))
		{
			_delay_ms(20);
			if (bit_is_clear(PIND, PD7))
			{
				usart_puts(mensaje);
				loop_until_bit_is_set(PIND, PD7);
			}
		}

		if (usart_kbhit())
		{
			char c = usart_getc();
			if (c == '0')
				toogle_bit(PORTD, PD6);
		}
	}
}
```

Un “poco” más legible, no?

ADC

El ADC o Conversor Analógico Digital es un módulo del microcontrolador que sirve para traducir una señal analógica a un valor digital.
La señal analógica es un nivel de tensión cuyo origen puede ser desde un divisor de tensión hasta el más complicado transductor de señal.

Resolución y referencia de tensión

La circuitería del ADC realiza la conversión por aproximaciones sucesivas dentro del rango de referencia de tensión y con una determinada resolución medida en bits.
La referencia de tención Vref+ puede estar asociada a VDD=5V y Vref-, que normalmente es 0V, mientras que la resolución viene dada por esta fórmula:

$$Resolucion = \frac{Vref+ - Vref-}{2^n - 1}$$

Donde 2^n es la cantidad de valores digitales posibles como resultado de la conversión, si n es 10 la resolución de voltaje del conversor será: $5/(2^{10}-1) = 4.88mV$

El valor obtenido por el ADC para una determinada tensión de entrada Vin será:

$$ValorADC = \frac{2^n - 1}{Vref+} * Vin$$

Configuración de registros

Las aproximaciones sucesivas de la conversión llevan tiempo, entre 50khz a 200khz para obtener la resolución máxima. Se pueden hacer más rápido pero se pierde resolución.

El tiempo de sampleo, en un ATmega8, se obtiene por prescalar el oscilador con el registro **ADCSRA** (bits **ADPSx**)

ADPS2	ADPS1	ADPS0	Divisor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8

1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

La selección del canal se logra con el registro **ADMUX** (bits **MUX3:0**), y la referencia de tensión con los bits **REFS1:0**:

REFS1	REFS0	VRef
0	0	AREF, referencia interna se apaga.
0	1	AVCC con condensador en el pin AREF
1	0	Reservado
1	1	Referencia interna de 2.56V con condensador en pin AREF

La conversión se inicia con un 1 en el bit **ADSC** del registro **ADCSRA** el cual se mantendrá en 1 mientras dure la conversión, al finalizar el valor queda en **ADCL** y **ADCH** con el formato que establece **ADLAR** (ADC left adjust result) del registro **ADMUX**.

Si ADLAR es **0** (ajustado a la derecha) el resultado en ADCL y ADCH se guarda así:

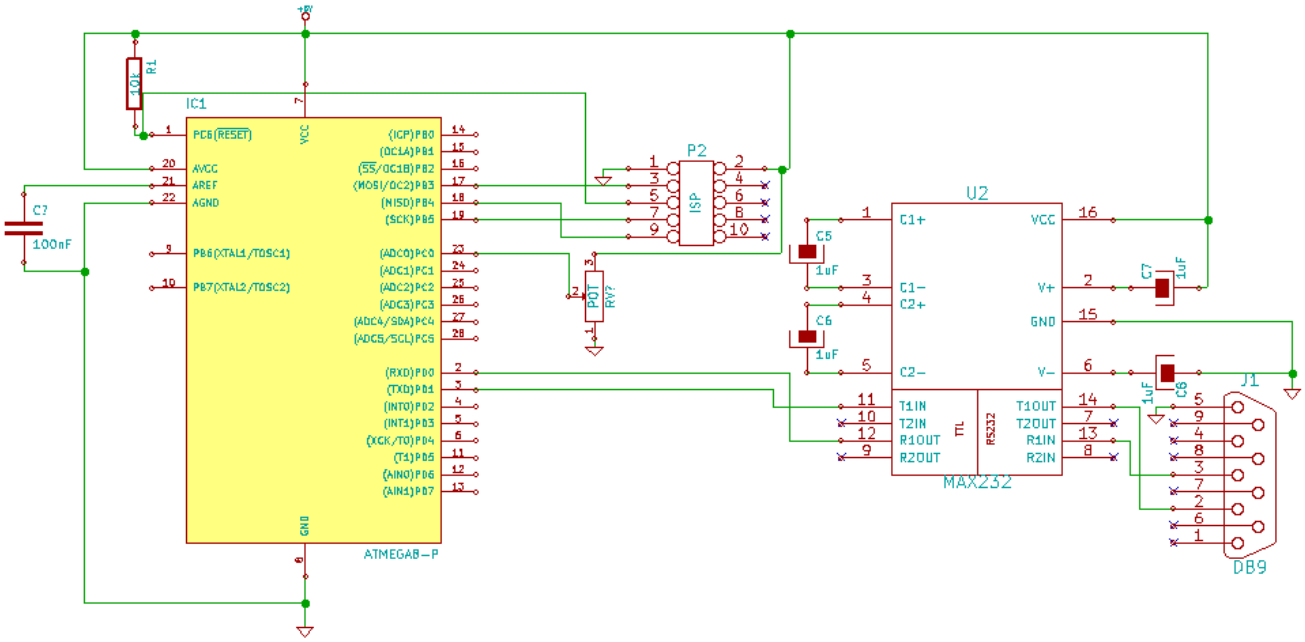
-	-	-	-	-	-	ADC9	ADC8	ADCH
ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL

Si ADLAR es **1** (ajustado a la izquierda):

ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
ADC1	ADC0	-	-	-	-	-	-	ADCL

Si el resultado está ajustado a la izquierda y no se requieren más de 8 bits, es suficiente con leer ADCH. De lo contrario, se debe leer primero ADCL y luego ADCH.

Circuito de prueba




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

#include "usart.h"

#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)

int main(void)
{
	usart_init(BAUD_PRESCALE);

	// ADC Enable, Prescaler=8 (1Mhz/8=125khz)
	ADCSRA = (1<<ADEN) | (1<<ADPS1) | (1<<ADPS0);
	ADMUX = 0;                                  // elige canal 0 ADC0 (pin PC0)
	ADMUX |= (1<<REFS0);                        // VRef = AVCC

	char buffer[10];

	for (;;)
	{
		ADCSRA |= (1<<ADSC);                    // comienza la conversión
		while (ADCSRA & (1<<ADSC))
		{
```

```
                }

                uint16_t valorADC = ADCL;           // 10 bits requeridos -> leemos primero ADCL
                valorADC |= (ADCH << 8);            // agregamos los dos bits de ADCH

                itoa(valorADC, buffer, 10);       // convertimos el valor decimal en vector de char

                usart_puts(buffer);               // enviamos el resultado de la conversión via USART
                usart_puts("\r\n");

                _delay_ms(50);
        }
}
```

Al arrancar la consola en la PC el microcontrolador enviará el valor digital resultado de la conversión de tensión de entrada en ADC0 (pin PC0). Si el lector nota cierta imprecisión puede deberse a ruidos en la alimentación o falta de condensadores de desacoplo. Cuando la resolución es mayor a 8 bits a veces es recomendable realizar 4 lecturas y calcular un promedio u otra técnica que logre estabilizar los valores leidos.

El módulo ADC utiliza mucho tiempo para la conversión, esto hace que la técnica usada de sampleo no sea eficiente ya que deja al CPU sin posibilidad de hacer otra tarea o simplemente quedar inactivo durante la conversión para bajar el consumo de energía.

A continuación estudiaremos las interrupciones y su utilidad para abordar estos problemas.

Interrupciones

Una interrupción es un evento externo (o interno) que pausa la ejecución actual del micro para completar una tarea **breve** y luego continuar donde había quedado.
Un ejemplo de interrupción es la detección de un flanco ascendente o descendente en un pin del micro.
Poder realizar otras tareas mientras el evento no ocurra libera al microcontrolador de preguntar en cada momento si dicho suceso ocurrió o no.

Consideraciones

- No hacer cálculos intensivos dentro de una interrupción
- No hacer largos bucles (loops).
- Codificar, en lo posible, pensando en la simultaneidad de interrupciones.
- Si se desea modificar una variable global esta debe estar definida como **volatile**. A veces incluso es necesario algún mecanismo de sincronización de acceso a variables.
- Mientras se está ejecutando la interrupción el programa principal está detenido.

Fuentes de interrupciones

Estas son las dos principales fuentes de interrupción:

1. Interrupción por hardware, que puede darse como respuesta a un cambio externo en un pin o por un timer.
2. Interrupción por software, causada por una o más acciones ejecutadas en el código principal.

Vectores de Interrupción
Esta es la lista de interrupciones de un ATmega8 ordenado por prioridad:

Origen	Definición
RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
INT0	External Interrupt Request 0
INT1	External Interrupt Request 1
TIMER2_COMP	Timer/Counter2 Compare Match
TIMER2_OVF	Timer/Counter2 Overflow
TIMER1_CAPT	Timer/Counter1 Capture Event

TIMER1_COMPA	Timer/Counter1 Compare Match A
TIMER1_COMPB	Timer/Counter1 Compare Match B
TIMER1_OVF	Timer/Counter1 Overflow
TIMER0_OVF	Timer/Counter0 Overflow
SPI, STC	Serial Transfer Complete
USART, RXC	USART, Rx Complete
USART, UDRE	USART Data Register Empty
USART, TXC	USART, Tx Complete
ADC	ADC Conversion Complete
EE_DRY	EEPROM Ready
ANA_COMP	Analog Comparator
TWI	Two-wire Serial Interface
SPM_RDY	Store Program Memory Ready

Manejando una interrupción

Para manejar una interrupción se debe:

- Habilitar el registro de interrupciones global (bit I de **SREG**) usando las funciones **sei**()
- Habilitar la interrupción a manejar en el registro correspondiente.
- Agregar una función con la macro **ISR**(nombre_de_vector) que contenga el código de la interrupción.

Ejemplo: USART con interrupciones

El lector recordará el ejemplo de comunicación serie, en el mencionado ejemplo el micro está constantemente consultando los registros de USART a la espera de un dato en el buffer o esperando que el buffer de transferencia se vacíe.
Lo que haremos a continuación es utilizar los vectores de USART mencionados antes, para liberar al micro de dichas tareas.

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define USART_BAUDRATE 4800
#define BAUD_PRESCALE (F_CPU / 16 / USART_BAUDRATE - 1)

int main(void)
{
	UCSRB |= (1 << RXEN) | (1 << TXEN);  // Inicializa la USART para envio y recepción.

	UCSRC |=  (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Asincrónica, 8,N,1

	UBRRH = (BAUD_PRESCALE >> 8);
	UBRRL = BAUD_PRESCALE;

	UCSRB |= (1 << RXCIE ); // Habilita la interrupción por recepción de dato USART completada

	sei(); // Habilita el flag de interrupciones globales

	for (;;)
	{
		// En este ejemplo el CPU está en espera
		// aquí se podría usar una técnica de
		// power-saving durmiendo el procesador
		// (ver en datasheet tema sleep).
	}
}

ISR(USART_RXC_vect)
{
	uint8_t datoLeido;

	datoLeido = UDR;
	UDR = datoLeido;
}
```

Timers

Un timer o temporizador, utilizado obviamente para medir tiempo, en un AVR sirve para hacer tareas de forma asíncrona al programa principal. Esto puede pensarse así debido a que un Timer funciona de forma independiente al núcleo de AVR.

Los timers pueden configurarse para producir una salida en un predeterminado pin reduciendo la carga del micro.

Un timer debe tener un reloj asociado que le diga el paso, cada pulso incrementa el timer en uno, el timer mide los intervalos en periodos de la siguiente forma:

$$Resolucion = \frac{1}{Frecuencia}$$

Esto significa que el menor tiempo posible que el timer puede medir es un periodo de la señal de reloj de entrada. Por ejemplo, si alimentamos al timer con una señal de 100Hz, el periodo sería:

$$Resolucion = \frac{1}{100Hz} = 0.1s$$

El periodo sería de 0.1 segundos, de modo que nuestro timer mediría en múltiplos de este. Si medimos un retardo de 60 periodos de timer, entonces el retardo total sería 60 veces 0.1 segundos o 0.6 segundos.

Timer sin prescaler

Veamos un ejemplo utilizando el circuito de USART, en este caso estamos con una frecuencia trabajo de 1 Mhz de oscilador interno, si deseamos medir por ejemplo 1/25 de segundo tendremos que calcular la cantidad de periodos que hay en ese retardo usando esta fórmula

$$Cuenta\ de\ Timer = retardo * F_{CPU} = \frac{1}{25} * 1000000 - 1 = 39999$$

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

int main(void)
{
	DDRD |= (1<<DDD6);		// Led en PD6 como salida

	TCCR1B |= (1 << CS10);		// configura timer1 con reloj=F_CPU / 1
```

```
	for (;;)
	{
		if (TCNT1 >= 39999)		// Ya pasaron 40ms?
		{
			PORTD ^= (1<<PD6);	// hace parpadear el led
			TCNT1 = 0;		// pone a 0 el timer
		}
	}
}
```

Timer con prescaler

En el ejemplo anterior el retardo no puede ser mayor debido a los 16bits del timer, si se necesitara un delay mayor lo que se puede hacer es establecer un prescaler para que divida el reloj en potencias de 2. La resolución con prescaler pasa a tener esta fórmula:

$$Resolucion = \frac{Prescaler}{Frecuencia}$$

Como trabajamos con 1 Mhz de F_CPU estas son las resoluciones que obtenemos:

Valor de prescaler	Resolución (a 1Mhz de F_CPU)
1	1us
8	8us
64	64us
256	256us
1024	1024us

La fórmula de la cuenta del timer pasaría a:

$$Cuenta\ de\ Timer = \frac{F_{CPU}}{prescaler * retardo} - 1$$

Digamos que el retardo deseado sea de 1Hz estos serían los los valores de cuenta para cada prescaler:

Valor de prescaler	Cuenta de timer
1	999999
8	125000
64	15624

256	3905.25
1024	975.5625

Usaremos prescaler = 64 con la cuenta en 15624, entonces el código quedaría así:

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

int main(void)
{
	DDRD |= (1<<DDD6);

	TCCR1B |= (1 << CS11) | (1 << CS10);

	for (;;)
	{
		if (TCNT1 >= 15624)
		{
			PORTD ^= (1<<PD6);
			TCNT1 = 0;
		}
	}
}
```

El ejecutar este código nuestro Led estará titilando con una frecuencia de 1Hz.

Modo CTC

Con el modo CTC (Clear on Timer Compare) el hardware nos hará la comparación que realizabamos de forma manual aquí:

```
		if (TCNT1 >= 15624)
```

Estar comparando todo el tiempo este registro es ineficiente, gasta ciclos y un tanto inexacto. El modo CTC la comparación se hace en el firmware basta con cargar el valor de comparación en **OCR1A**, hace **1** el bit **WGM12** de **TCCR1B** comparando luego el bit **OCF1A** de **TIFR** con **1.**

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

int main(void)
{
	DDRD |= (1<<DDD6);                                  // Led en PD6 como salida

	TCCR1B |= (1 << WGM12);                             //Modo CTC
	OCR1A = 15624;                                      //valor de comparación

	TCCR1B |= (1 << CS11) | (1 << CS10);  // configura timer1 con reloj=F_CPU / 64
```

```
        for (;;)
        {
                if (TIFR & (1 << OCF1A))                    // Ya pasó 1s?
                {
                        PORTD ^= (1<<PD6);                  // hace parpadear el led

                        TIFR = (1 << OCF1A);                //borra el flag CTC
                }
        }
}
```

Modo CTC con interrupciones

Hacemos lo mismo que en modo CTC pero usando interrupciones sin necesidad de estar consultando por el flag, usamos la interrupción **TIMER1_COMPA** habilitando el bit OCIE1A de **TIMSK**.

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main(void)
{
        DDRD |= (1<<DDD6);                                  // Led en PD6 como salida

        TCCR1B |= (1 << WGM12);                             //Modo CTC
        OCR1A = 15624;                                      //valor de comparación

        TIMSK |= (1 << OCIE1A);                             // Habilita las interrupciones CTC
        sei();                                              // activa las interrupciones

        TCCR1B |= (1 << CS11) | (1 << CS10);  // configura timer1 con reloj=F_CPU / 64

        for (;;)
        {
        }
}

ISR (TIMER1_COMPA_vect)
{
        PORTD ^= (1<<PD6);                          // hace parpadear el led
}
```

Modo CTC - Output Compare

Es posible optimizar aun más nuestro ejemplo anterior colocando el led y la resistencia en PB1 (OC1A) e indicandole al micro que en el momento en que se detecte una igualdad en la comparación alterne la salida OC1A logramos que el hardware nos haga titilar el led!

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

int main(void)
{
	DDRB |= (1<<DDB1);                                  // Led en PB1 como salida

	TCCR1B |= (1 << WGM12);                             //Modo CTC
	OCR1A = 15624;                                      //valor de comparación

	TCCR1A |= (1 << COM1A0);                            // output compare canal A

	TCCR1B |= (1 << CS11) | (1 << CS10);  // configura timer1 con reloj=F_CPU / 64

	for (;;)
	{
	}
}
```

Esta claro que esta salida puede ser usada para generar una señal cuadrada en OC1A/B de una frecuencia calculada usando las fórmulas ya aprendidas ;)

De forma similar al Modo CTC con interrupciones se podría habilitar usar el modo normal u overflow donde el contador no se reinicia y se produce un desbordamiento del contador de 8 o 16 bits afectando al bit **TOVx** del registro **TIFR**.

PWM

El PWM o modulación por ancho de pulso se lo puede entender como otro funcionamiento de un timer. El objetivo es la generación de una forma de onda cuadrada de ancho regulable.

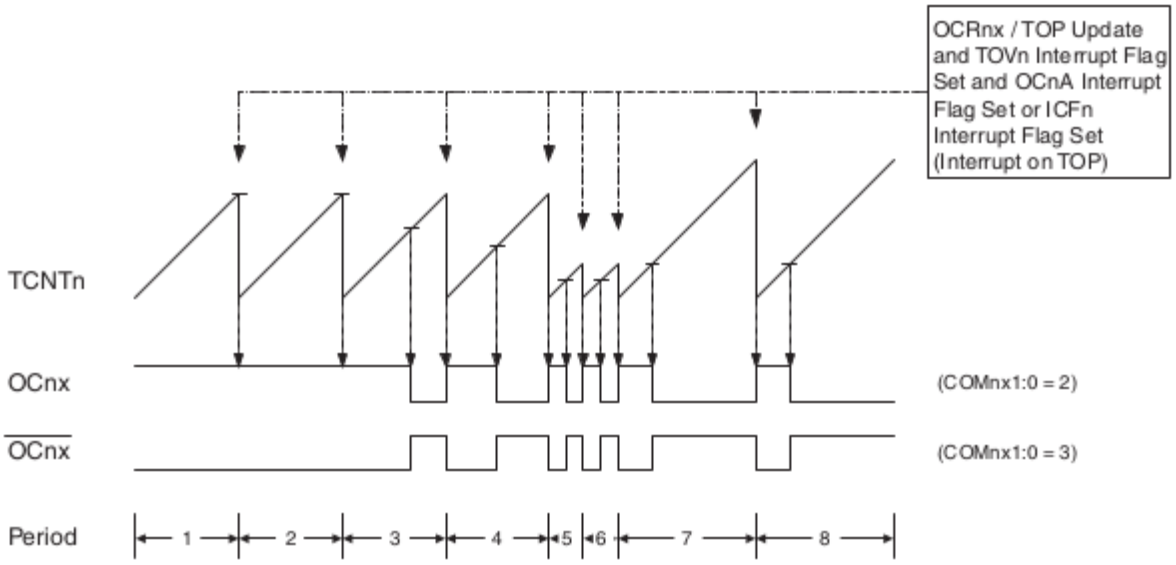
Modo Fast-PWM

Fórmula de PWM resolution

$$Resolucion = \frac{log(TOP+1)}{log(2)}$$

Fórmula de la frecuencia del PWM

$$F_{OCnx}PWM = \frac{F_{CPU}}{N*(1+TOP)} con\ prescaler\ N(1{,}8{,}256\,o\,1024)$$




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

int main(void)
{
	DDRB |= (1<<DDB1);				// Led en PB1 como salida

	TCCR1A |= (1 << COM1A1);			// borra OC1A en comparación
	TCCR1B |= (1 << WGM13) | (1 << CS11);		// modo PWM, prescaler=8

	ICR1 = 200;			// fpwm = 1/(2*200) = 2.5khz
	OCR1A = 50;			// duty = 25%

	for (;;)
	{
	}
}
```

Registros de PWM

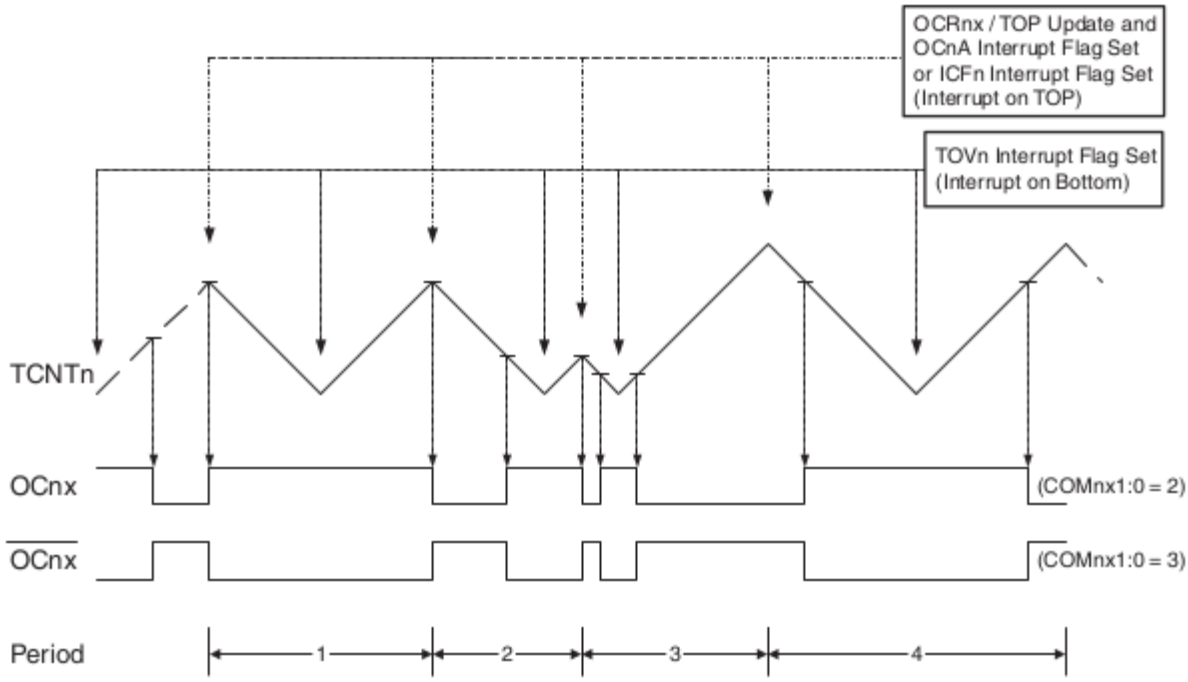
Estos son los registros que afectan el funcionamiento del módulo de PWM del microcontrolador.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Modo Phase Correct

Fórmula de la frecuencia del PWM

$$F_{OCnx}PCPWM = \frac{F_{CPU}}{2*N*TOP} con\ prescaler\ N(1,8,256\,o\,1024)$$

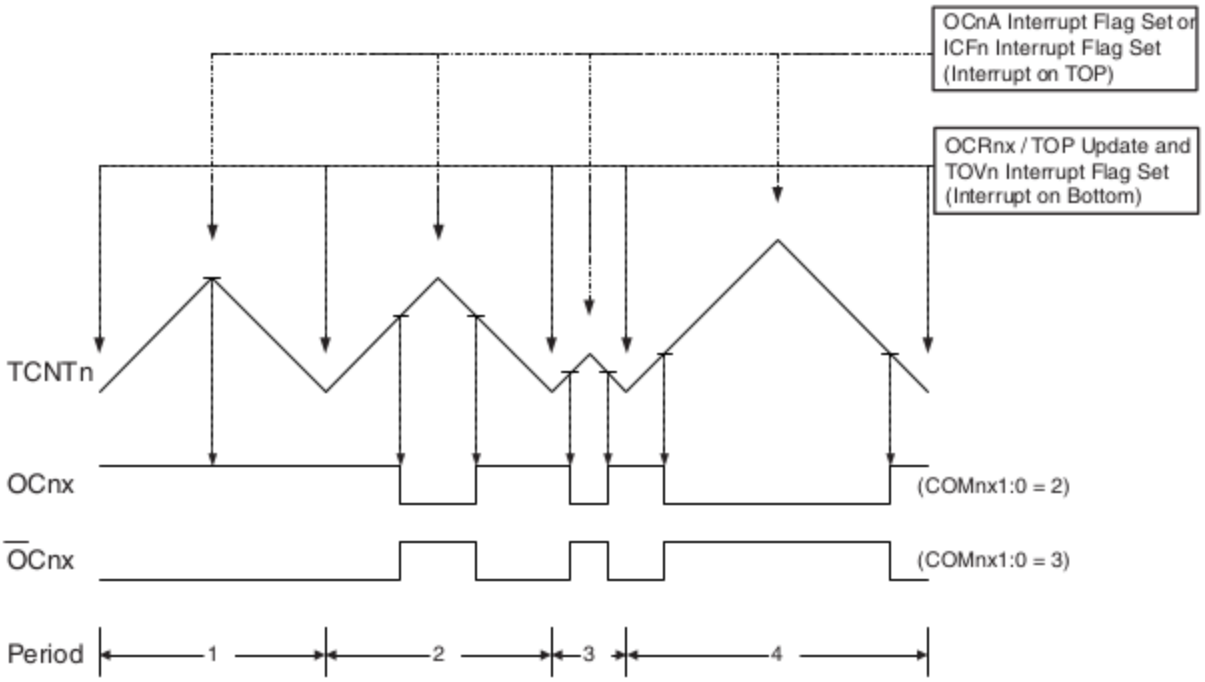


En este modo el timer cuenta desde 0 hasta el TOP y luego baja desde el valor TOP hasta 0. Debido a esta doble operación la frecuencia de la señal es inferior a Fast PWM pero posee el beneficio de generar una onda simétrica.

Modo Phase and Frecuency Correct

Fórmula de la frecuencia del PWM

$$F_{OCnx}PFCPWM = \frac{F_{CPU}}{2*N*TOP} con\ prescaler\ N(1{,}8{,}256\,o\,1024)$$



En este modo la onda queda centrada en el periodo y es simétrica en todos sus periodos.

EEPROM

Rutinas comunes

Definidas en <avr/eeprom.h> para facilitar el acceso a esta memoria

```
uint8_t eeprom_read_byte (const uint8_t *__p) __ATTR_PURE__;
uint16_t eeprom_read_word (const uint16_t *__p) __ATTR_PURE__;
uint32_t eeprom_read_dword (const uint32_t *__p) __ATTR_PURE__;
float eeprom_read_float (const float *__p) __ATTR_PURE__;

void eeprom_write_byte (uint8_t *__p, uint8_t __value);
void eeprom_write_word (uint16_t *__p, uint16_t __value);
void eeprom_write_dword (uint32_t *__p, uint32_t __value);
void eeprom_write_float (float *__p, float __value);
```

Ejemplo de uso

```
uint8_t dato;
dato = eeprom_read_byte(( uint8_t *) 32) ;
```

Acceso por bloque

```
void eeprom_write_block (const void *__src, void *__dst, size_t __n);
void eeprom_read_block (void *__dst, const void *__src, size_t __n);
```

Modificador EEMEM

```
#define EEMEM __attribute__((section(".eeprom")))
```

```
#include <avr/eeprom.h>

uint8_t EEMEM caracter;
uint16_t EEMEM entero;
uint8_t EEMEM cadena[10];

int main ( void )
{
        uint8_t caracterSRAM;
```

```
        uint16_t enteroSRAM;
        uint8_t cadenaSRAM[10];

        caracterSRAM = eeprom_read_byte(&caracter);
        enteroSRAM = eeprom_read_word(&entero);
        eeprom_read_block((void*) &cadenaSRAM, (const void*) &cadena, 10);
}
```

Estableciendo valores iniciales

```
uint8_t EEMEM variable = 45;
```