

# ESTUDIO TEÓRICO DEL MICRO ATMEGA328P EN ARDUINO UNO

El propósito de estos apuntes no es el estudio exhaustivo de Arduino, sino simplemente dar una visión general de sus posibilidades. Todos los temas que se tratan en este tutorial son muy avanzados. No son imposibles, desde luego, pero sí que se necesitan conocimientos previos sobre el manejo de la placa para entenderlos.

Sin embargo, son muy interesantes y nos ayudarían a comprender los límites y las potencialidades de nuestros dispositivos.

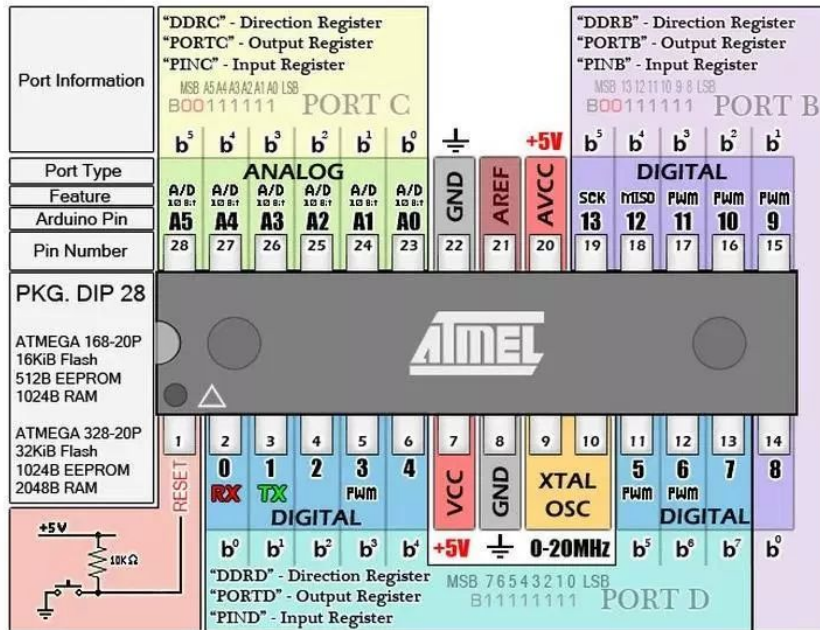
Entre los temas a tratar en estos apuntes encontrarás: manejo de puertos, interrupciones por hardware y timers, multitarea, el problema del rollover y una introducción a la programación orientada a objetos (POO).



por [Aurelio Gallardo](#)



# De pines multipropósito, puertos y registros



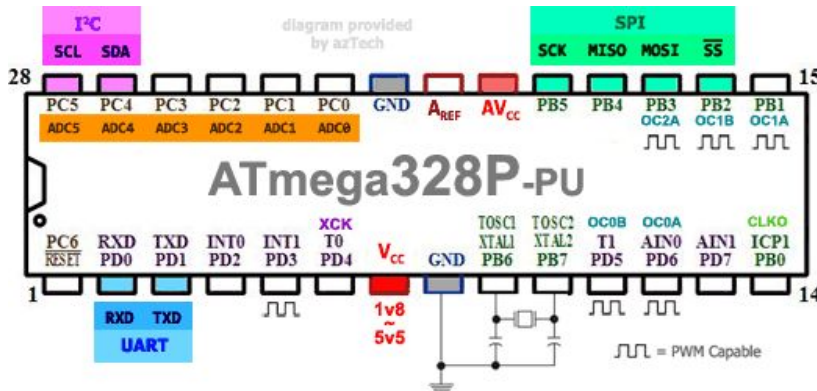
ATMEL - ATMEGA 168/328 - 20P  
Robin Farrell 2010

No es la intención de estos apuntes conocer detalladamente los vericuetos de un microprocesador ATmega, el cerebro de nuestros arduinos, pero sí acceder o manejar directamente algunas de sus funcionalidades.

Cada pin de nuestro ATmega puede tener una o más de una función.

<https://aprendiendoarduino.wordpress.com/2017/09/03/puertos-digitales-arduino-avanzado/>

# De pines multipropósito, puertos y registros



**Registro:** son memorias de muy baja capacidad pero acceso muy rápido. Arduino posee varias: de ellas nueve, tres asociadas a cada puerto.

Por ejemplo, del pin 28 del ATmega328 (Arduino Uno), podemos decir:

- Es el sexto pin más significativo del puerto C (nº 5 → PC5)
- Está conectado al conversor analógico digital. Luego se puede configurar como entrada analógica (A5)
- El protocolo I<sup>2</sup>C lo utiliza como señal SCL.

La mayor parte de los pines del microprocesador se agrupan en tres “zonas”. Cada zona es un puerto: **PORTB**, **PORTC** y **PORTD** y cada puerto lo controla Arduino con tres registros: **DDR**, **PORT** y **PIN**.

## De pines multipropósito, puertos y registros

Así, por ejemplo, el puerto B de Arduino (los pines del 8 al 13) , se controlan con tres registros:

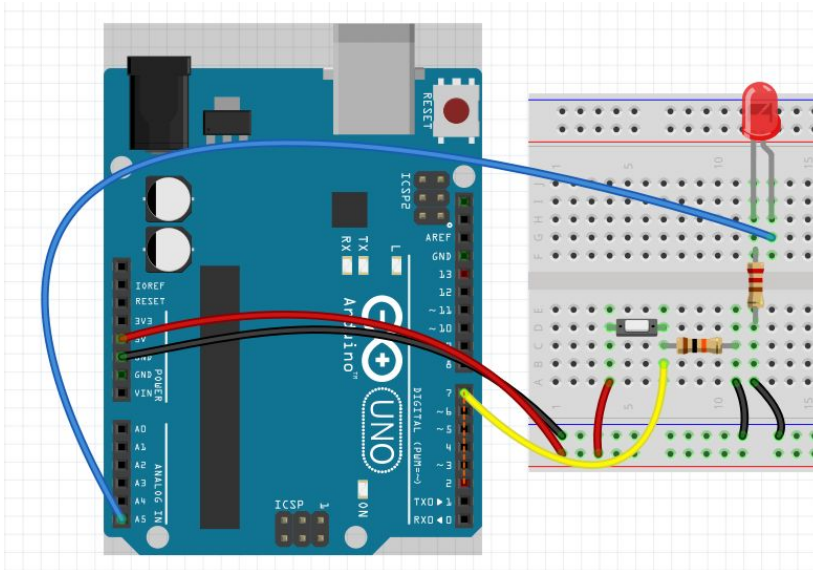
- DDRB, determina si el pin es una entrada (0) o una salida (1).
- PORTB, controla si el pin está en nivel alto (1) o en nivel bajo (0).
- PINB permite leer (sólo lectura) el estado de un pin.

Todos los registros son de 8 bits. Pero hay algunas limitaciones para usarlos ya que:

- En el puerto B los bits 6 y 7 controlarían los pines del ATmega328 nº 9 y 10, pero en un Arduino están conectados a un cristal oscilador externo. No pueden usarse. Así que el puerto B en un Arduino debe quedar como **00XXXXXX**.
- En el puerto C, los pines 6 y 7 tampoco se pueden usar. El 6 está conectado al RESET y el 7 en el ATmega328 no está cableado. Asimismo el puerto C es **00XXXXXX**. Con él podemos controlar los pines A0-A5.
- Los pines de Arduino del 0 al 7 sí son los ocho bits del puerto D. Pero el 0 (RX) y el 1 (TX) son los pines de comunicación USB. Podemos usarlos siempre y cuando no interfieran en la programación del Arduino. En principio, **XXXXXXXX**.

## Ejemplo: enciendo un led con un pulsador, como siempre

Mucha palabrería: al grano, primero hacemos un programa “como siempre” que encienda un LED al pulsar un botón.



```
int salida = 19;
int entrada = 7;

void setup() {
  pinMode(salida, OUTPUT);
  pinMode(entrada, INPUT);
}

void loop() {
  digitalWrite(salida, digitalRead(entrada));
  delay(15);
}
```

¿No sabíais que los pines A0... A5 son las salidas digitales D14...D19? ¡¡Pues ya lo sabéis!!



## Ejemplo: enciendo un led con un pulsador, como nunca

Y segundo, un programa “como nunca” que encienda un LED al pulsar un botón.

```
/*
 *   Puertos:  Bit más significativo .. Bit menos significativo  (76543210)
 */


boolean pulsado = 0;

void setup() {
  DDRD=B01111111; // puerto 7 como entrada.
  //Los demás como salidas (aunque no conecto nada en ellos).
  //Puerto D, los pines del 0 al 7
  DDRC=B00100000; // puerto 5 (A5) como salida.
  // 6 y 7 no cuentan y los demás como entradas
  Serial.begin(9600);
}

void loop() {
  // Voy a leer el registro en la posición 7 del DDRD
  // A) el número 1 (00000001) lo desplazo 7 posiciones a la izquierda (1<<7)
  // con lo que obtengo el (10000000)
  // B) esa operación la comparo con el registro de lectura PIND. Le hago un AND
  // Si es 1 es que está pulsado y 0 apagado
  pulsado = PIND & (1<<7);
  // La condición de pulsado la traslado al quinto lugar
  // del registro C para que lo escriba
  PORTC=(pulsado<<5);
  Serial.println(pulsado);
  delay(15);
}
```

## Ejemplo: mucho cuento...

Que “mucho rollo”... Que con ésto es suficiente (3 líneas de código).



```
void setup() {  
    DDRD=B01111111; // puerto 7 como entrada.  
    DDRC=B00100000; // puerto 5 (A5) como salida.  
}  
  
void loop() {  
    // La lectura del puerto D en el valor 7  
    // por si acaso con AND 1  
    // La traslado al quinto lugar del PORTC  
    PORTC=((PIND>>7 & 1)<<5);  
}
```

### Ventajas:

- Cada instrucción necesita uno o varios ciclos de reloj. Evidentemente definir así los pines necesitan muchos menos que una combinación de pinMode, digitalWrite, etc.
- Los pines se cambian simultáneamente. No son necesarios bucles.
- Se puede ahorrar en memoria de programa con este método.

### Inconvenientes

- La programación es más difícil, ya que estamos a nivel de la máquina. Es más fácil equivocarse.
- No es recomendable para novatos.



## Ejemplo: más ejemplos

```
// Ejemplo con pulsador en PIN7
// LED en A5
byte antes = 0;

void setup() {
  DDRD=B01111111; // puerto 7 como entrada.
  DDRC=B00100000; // puerto 5 (A5) como salida.
}

void loop() {
  PORTC=( ( (PINC>>5) xor ((PIND>>7)>(antes>>7)) ) <<5);
  antes = PIND & B10000000;
  delay(50);
}
```

---

```
// Luces coche Fantástico
// 4 LEDS como salidas en los pines
// 4,5,6, y 7. Puerto D
```

```
unsigned tiempo = 0;
```

```
void setup() {
  DDRD=B11111111; // puerto 4,5,6,7 como salida.
  /// Las demás también salidas pero no las voy a usar
}
```

```
void loop() {
  tiempo = millis()/100; // A menos valor, más rápido
  // Control de puertos y tiempo avanzado.
  // tiempo%4 --> señal periódica 0 1 2 3 -->
  // Fórmula ciclo positivo --> x
  // Fórmula ciclo negativo --> 3-x
  // Ciclo positivo en tiempo/4 resto entre 2 igual a cero
  // Ciclo negativo en tiempo/4 resto entre 2 igual a uno
  // El último suma 4, porque empiezo en el pin 4...
  PORTD = (1<< ((3*((tiempo/4)%2==1)-(tiempo/4)*((tiempo/4)%2==1)+(tiempo/4)*((tiempo/4)%2==0))+4) );
}
```



# Interrupciones de hardware



Imaginaros que estamos en clase, explicando, y de pronto llama el director o directora a la puerta. Te callas, claro. Comenta no sé qué de una excursión, les da un papel a los chavales, da las gracias y se marcha. Y tú sigues exactamente por donde lo dejaste.

Esta misma situación, idéntica, nos la encontramos en los procesadores. Los procesadores ejecutan un programa, pero, además pueden estar a la escucha en alguno de sus pines de ciertas señales. Si detectan una señal en un pin que escucha una interrupción, paran inmediatamente el flujo del programa principal, actúan según una rutina asociada a la interrupción (se llama *ISR* -Interruption Service Rutine- , del tipo callback -respuesta-) y al terminarla vuelven al punto del programa donde lo habían dejado.

<https://www.luisllamas.es/que-son-y-como-usar-interrupciones-en-arduino/>

# Interrupciones de hardware

Las interrupciones tienen alguna ventaja: no hay que esperar hasta el punto del programa en el que escucho, por ejemplo, el estado de un botón por lo que no tengo el riesgo de “no haberlo escuchado”; no consumen por tanto energía en estar continuamente “escuchando” esa entrada.

Las interrupciones son de dos tipos: **por hardware o por tiempo** (timers). Dentro de las primeras, que son las que veremos, hay cuatro eventos que pueden dispararlas: **RISING** (flanco de subida:  $0 \rightarrow 1$ ), **FALLING** (flanco de bajada:  $1 \rightarrow 0$ ), **CHANGING** (rising+falling) y **LOW** (mientras esté en estado bajo).

Arduino Uno puede escuchar interrupciones en **sus pines 2 y 3**.

Respecto a las ISR:

1. **No se pueden ejecutar dos a la vez**. Si acaso, se ejecuta una después de otra.
2. **No recibe nada y no devuelve nada**.
3. **Deben ser cortas**. Su uso prolongado afecta al control del tiempo en el proceso principal. Frecuentemente se usan para activar un flag, incrementar un contador, o modificar una variable. Esta modificación será atendida posteriormente en el hilo principal, cuando sea oportuno.
4. Las variables que use la interrupción que después se usen en el programa principal **deben declararse como “volatile”**. Esto fuerza al procesador a comprobarlas antes de usarlas y así comprobar si un proceso externo las ha modificado.

# Interrupciones de hardware: ejemplo

5. Si ejecuto una ISR, en el programa principal *las funciones millis() y micros() no se actualizan*, ya que éstas dependen de interrupciones timers que se detienen. Por eso, no es conveniente usar muchas ni que éstas sean muy largas, por la distorsión que podemos tener en la cuenta del tiempo.
6. Dentro de las ISR: millis() no funciona y delay() tampoco; micros() funciona durante 500us y delayMicroseconds() sólo durante ese período.

**Ejemplo:** usaré una modificación del último programa realizado para hacer un juego de luces mediante puertos. Un pulsador, asociado a una interrupción, cambiará el timing de las luces, haciéndolas cada vez más lentas.

```
// Luces coche Fantástico: 4 LEDS como salidas en los pines 4,5,6, y 7. Puerto D
// Interrupción en el pin2: Colocar un pulsador en este pin

unsigned tiempo = 0;
volatile int frecuencia = 100;

void setup() {
  DDRD=B11111111; // puerto 4,5,6,7 como salida.
  /// Las demás también salidas pero no las voy a usar
  attachInterrupt(digitalPinToInterrupt(2), cambioFrecuencia, FALLING);
  Serial.begin(9600);
}

void loop() {
  tiempo = millis()/frecuencia; // A menos valor, más rápido
  PORTD = (1<< ((3*((tiempo/4)%2==1)-(tiempo%4)*((tiempo/4)%2==1)+(tiempo%4)*((tiempo/4)%2==0))+4) );
  Serial.println(frecuencia);
}

void cambioFrecuencia() {
  frecuencia+=10;
}
```

Sería conveniente leer también este artículo de Luis Llamas sobre el efecto rebote o “debounce”:  
<https://www.luisllamas.es/debounce-interrupciones-arduino/>

# Interrupciones de hardware: órdenes

Las órdenes usadas con las interrupciones son:

- **attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)** → asocio una interrupción a un PIN, defino la función ISR y escojo un modo.
- **detachInterrupt(digitalPinToInterrupt(pin))** → desactivo una interrupción
- **NoInterrupts()**, desactiva la ejecución de interrupciones hasta nuevo orden. Equivale a **cli()**
- **Interrupts()**, reactiva las interrupciones. Equivale a **sei()**

# Introducción a la multitarea

```
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

En el de la derecha, controlo dos LEDs a distinta frecuencia. El uso inteligente de la función `millis()` permite realizar una tarea sin “acaparar” el microprocesador para otras.

Fijémonos en el programa BLINK, el primero que solemos poner cuando aprendemos.

En este programa, mientras se ejecuta el `delay` no podemos hacer nada más. Sólo podemos controlar bien una tarea, no más de una.

// Parpadeo dos LEDs MULTITAREA

```
unsigned tiempo = 0;
int frecuencial = 1000; // cada segundo
int frecuencia2 = 300; // cada 0.3 s

void setup() {
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
}

void loop() {
  tiempo = millis(); // cada segundo
  digitalWrite(6, (tiempo/frecuencial)%2);
  digitalWrite(7, (tiempo/frecuencia2)%2);
}
```

El fantástico tutorial de Luis Llamas nos explica muy bien métodos generales de conseguir la multitarea, incluyendo el uso de librerías específicas:

<https://www.luisllamas.es/multitarea-en-arduino-blink-sin-delay/>

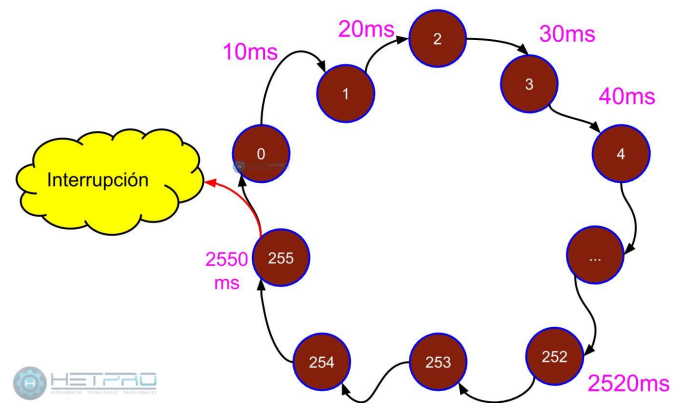
# Timers

¿Os acordáis de las interrupciones de hardware? Eran señales externas que activábamos y que permitían realizar una función de forma urgente, parando momentáneamente el programa principal.

Arduino UNO posee un oscilador de 16MHz y un circuito independiente con tres registros que realizan una cuenta. El primer registro timer0 es de 8 bits, el segundo timer1 de 16 bits y el tercero timer2 de 8 bits. A cada uno puedo asociar una frecuencia (dividida del reloj principal) inferior a 16MHz. Supón que al timer0 le asocio 100Hz. Acabaría su cuenta al cabo de  $255 \times 10\text{ms} = 2.55\text{s}$ . En ese momento genera una señal, una interrupción, que puede ser leída por nuestro microprocesador.

- <https://hetpro-store.com/TUTORIALES/arduino-timer/>
- <https://creatividadcodificada.com/arduino/timer-con-arduino-o-interrupciones-internas/>
- <https://www.electrontools.com/Home/WP/2016/05/13/como-usar-las-interrupciones-en-arduino/>

Timer-0 8-bits programado a 100Hz en incremento.



# Timers

Pero nuestro Arduino **YA USA** los timers de forma predeterminada. El timer0 lo usa para las funciones `millis()`, `micros()` y `delay()` así que mejor no lo tocamos. El timer1 para funciones relacionadas con los servos y el timer2 para la función `tone()`.

Así que si nos atrevemos a personalizar los timers, usaremos el timer2 (mientras no genere notas musicales en el programa... :-)).

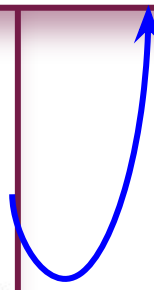
```
// Usando timer2. El timer2 es de 8bits
// usa por defecto divisiones del reloj 16MHz.
// La primera que usa es la división 1/2 --> 8MHz.
// A partir de ahí, 1MHz, 250Hz, etc. verlo en la web
// https://hetpro-store.com/TUTORIALES/arduino-timer/

volatile int cuenta=0;
volatile boolean ESTADO = 0;

void setup() {
  pinMode(6,OUTPUT); // poner un led en el pin 6
  // A) Desactivo el uso de interrupciones. Registro SREG
  // 0 en el bit 7.
  SREG = (SREG & 0b01111111);
  // B) Limpio el registro contador del timer2
  TCNT2 = 0;
  // C) Habilitar la generación de una señal por desbordamiento
  // en el paso de 255 a 0. Bandera 0 del registro TIMSK2
  TIMSK2 =TIMSK2|0b00000001;
  // D) Registro TCCR2B. Sus tres bits menos significativos
  // CS20, CS21 y CS22 me dan la frecuencia del timer2. Por ejemplo
  // si tengo 111 --> 7812.5Hz o un período de 1.28e-4 seg = 0.128ms
  // cuando acabe la cuenta de 255 han pasado 32.6ms
  TCCR2B = 0b00000111; // o sea, funciona a 7812.5Hz
  // E) Habilito las interrupciones globales
  SREG = (SREG & 0b01111111) | 0b10000000; //Habilitar interrupciones
}
```

```
void loop() {
  // nada
}

// Después del Loop
ISR(TIMER2_OVF_vect){
  cuenta++; // contando a 30 permite
  // que el período sea ~ un segundo
  // 30 * 32.6ms = 978ms
  if(cuenta > 29) {
    digitalWrite(6,ESTADO);
    ESTADO = !ESTADO;
    cuenta=0;
  }
}
```





# Rollover o el overflow de millis() y micros()

Ya hemos aprendido que las funciones [millis\(\)](#) y [micros\(\)](#) se basan en señales que obtienen del registro [timer0](#) nuestro micro ATmega328 en Arduino. Ambas son contadores que suman 1 cada milisegundo y cada microsegundo respectivamente.

Estos contadores son del tipo de datos **long**. De hecho el tipo **long** son números de 4 bytes o 32 bits, de los cuales 1 reserva para el signo. Así que puede representar desde el número 0 al  $2^{31}-1=2.147.483.647$  para los positivos y del número  $-2^{31}=-2.147.483.648$  al -1 para los negativos.

Pero el tiempo no es negativo... ¿verdad? 🤔. Así que mejor que ser de tipo long son de un tipo parecido llamado **unsigned long** (largo sin signo) y los 32 bits los dedicamos a números positivos, así que su rango alcanza desde el 0 al  $2^{32}-1= 4294967295$ .

Si hacemos algunos cálculos, nos daremos cuenta que si contamos microsegundos, llegamos a ese número al cabo de unos 71.5 minutos y si contamos milisegundos al cabo de unos 49.7 días. Al término de ese tiempo, al llegar la cuenta al 4294967295, vuelve a cero (desborda - overflow) y si nuestro programa depende críticamente de ese valor de tiempo podemos tener un problema. Imagina que construyes un reloj y la cuenta del tiempo la obtienes de la función `millis()`: cuando llegues a ese punto, si no lo remedias, el reloj se desfazará.

Y no puedes evitarlo. Lo que tienes que hacer es programar teniéndolo en cuenta. Aquí te dejo un par de enlaces donde explican formas de evitarlo (o apoyarnos en circuitos como el [DS3231](#))

1. <https://www.norwegiancreations.com/2018/10/arduino-tutorial-avoiding-the-overflow-issue-when-using-millis-and-micros/>
2. <https://arduino.stackexchange.com/questions/12587/how-can-i-handle-the-millis-rollover/12588#12588> y <https://gastack.mx/arduino/12587/how-can-i-handle-the-millis-rollover>

# Introducción a la programación orientada a objetos (PPO)

La programación normal que llevamos a cabo con el IDE de Arduino se enmarca dentro de un paradigma imperativo: las órdenes se suceden unas detrás de otras, y se ejecutan en el orden en el que están escritas (y en el loop se hacen cíclicas). Corresponde a un modelo algorítmico.

Pero hay otros paradigmas. El que vamos a ver es el paradigma **de orientación a objetos**, implementable en el IDE de Arduino. No tenemos variables que obtienen un cálculo y en función de ellas se presentan varias salidas, sino que pensamos en los objetos que podemos representar en nuestro programa, sus propiedades y cómo podemos actuar sobre ellos modificando esas propiedades.

Por ejemplo, un LED. Un LED es un objeto que tiene dos **propiedades** o **atributos**: brilla y tiene un color. La ausencia de la propiedad brillo indica un estado del LED (apagado) y su presencia, el estado encendido. Su color es una propiedad fija inmodificable por software (hablamos de LEDs normales, no los tricolor).

Todos los LEDs que existen comparten esas propiedades. Así que **cuando me refiera a LED en abstracto me refiero no a un objeto sino a una clase (conjunto) de dicho objeto**. **Cuando me refiera a un LED concreto, una unidad concreta que pincho en la placa protoboard, tendré una instancia de ese objeto**.

Y ahora que lo pienso, pues un LED también puede conectarse. Así que el número del pin donde se conecta a nuestra placa también puede ser otra propiedad.

# Introducción a la programación orientada a objetos (PPO)

¿Y qué puedo hacer con un LED? Así de primeras se me viene tres acciones a la mente: apagarlo, encenderlo o hacerlo parpadear. Cada acción es un **método** posible.

```
// P00 objetos
// Ejemplo con 4 leds en los pines 4,5,6 y 7
// Lo primero que hago es definir una clase
class LEDS
{
    private: // atributos internos de la clase
        boolean estado;
        void aplicarEstado(){ // método privado
            digitalWrite(pin,estado);
        }

    public:
        int pin; // atributo externo de clase, accesible desde el programa principal
        LEDS(){ // constructor. Método que se llama IGUAL que la clase y se ejecuta
            // al instanciar un objeto.
            estado=LOW;
            pin=6; // al inicio tiene el 6
        }
        void apagar(){
            estado = LOW;
            aplicarEstado(); // método de apagado
        }
        void encender(){
            estado = HIGH;
            aplicarEstado(); // método de encendido
        }
};

LEDS l1,l2; // ambos se instancias apagados en pin 6
```

**Métodos y atributos privados. No son accesibles fuera de la clase.**

**Métodos y atributos públicos. Sí son accesibles fuera de la clase.**

**Creo dos instancias (objetos) de la clase LED**

# Introducción a la programación orientada a objetos (PPO)

```
// *****  
// SETUP  
// *****  
void setup() {  
  DDRD=B11111111;  
  l2.pin=7; // A uno tengo que cambiar a pin 7.  
}
```

**l1 y l2 se instanciaron con pin 6.  
A uno por lo menos le tengo que  
poner otro pin. l2.pin=7**

```
// *****  
// LOOP  
// *****  
void loop() {  
  l1.encender();  
  l2.apagar();  
  delay(200);  
  l2.encender();  
  l1.apagar();  
  delay(200);  
}
```

**En el LOOP se llaman los métodos de cada objeto.**

**Inténtalo añadiendo un  
método público más a la  
clase...**

**Y reescribiendo  
el LOOP**

```
void parpadear(int frecuencia){  
  if ((millis()/frecuencia)%2==0) {encender();}  
  if ((millis()/frecuencia)%2==1) {apagar();}  
}
```

```
// *****  
// LOOP  
// *****  
void loop() {  
  l1.parpadear(300);  
  l2.parpadear(1500);  
}
```