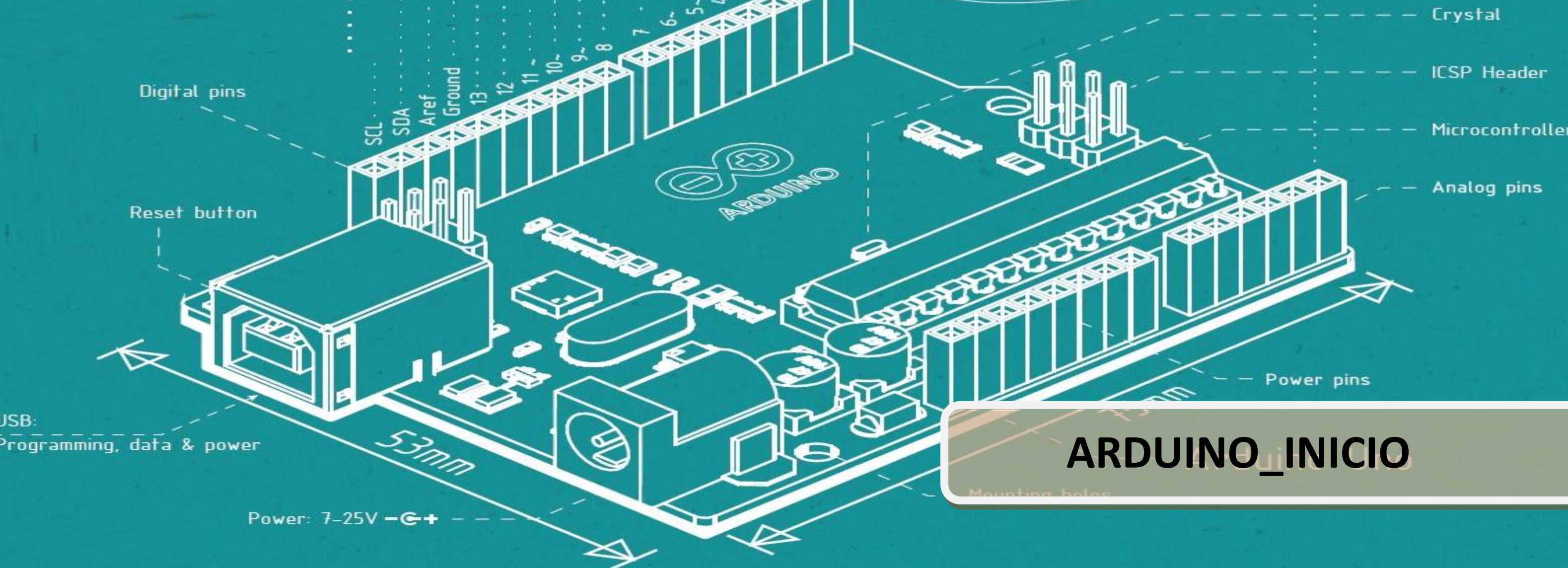


1. INTRODUCCIÓN A ARDUINO.

1.1 ¿Qué es Arduino?

Arduino es una plataforma de desarrollo de computación física (physical computing) de código abierto, basada en una placa con un sencillo microcontrolador y un entorno de desarrollo para crear software (programas) para la placa.

Puedes usar Arduino para crear objetos interactivos, leyendo datos de una gran variedad de interruptores y sensores y controlar multitud de tipos de luces, motores y otros actuadores físicos. Los proyectos con Arduino pueden ser autónomos o comunicarse con un programa (software) que se ejecute en tu ordenador. La placa puedes montarla tú mismo o comprarla ya lista para usar, y el software de desarrollo es abierto y lo puedes descargar gratis desde la página *www.arduino.cc/en/*.

El Arduino puede ser alimentado a través de la conexión USB o con una fuente de alimentación externa. La fuente de alimentación se selecciona automáticamente.

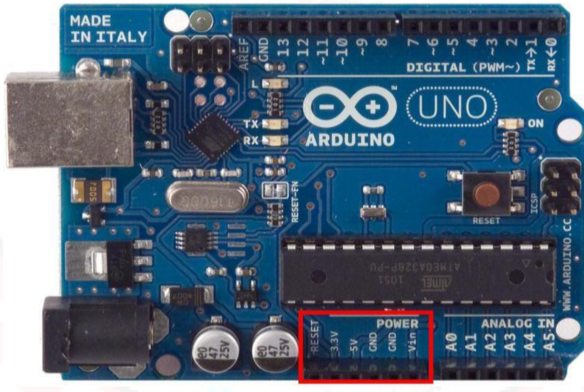
1.2 Hardware y cable USB

1.3 Especificaciones técnicas

Microcontroller	ATmega328
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current for I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328)
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz

1.4 Power, Inputs and Outputs.

1.4.1 Pines de alimentation (Power Pins)

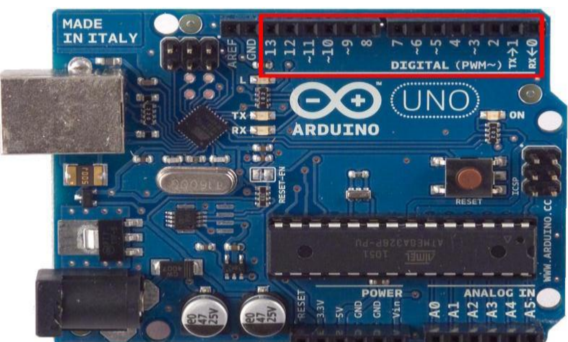


Bien alimentemos al arduino mediante la conexión USB o mediante una fuente externa (recomendada de 7-12V), vamos a tener unas salidas de tensión continua debido a unos reguladores de tensión y condensadores de estabilización. Estos pines son:

- VIN: se trata de la fuente tensión de entrada que contendrá la tensión a la que estamos alimentando al Arduino mediante la fuente externa.
- 5V: fuente de tensión regulada de 5V, esta tensión puede venir ya sea de pin VIN a través de un regulador interno, o se suministra a través de USB o de otra fuente de 5V regulada.
- 3.3V: fuente de 3.3 voltios generados por el regulador interno con un consumo máximo de corriente de 50mA.
- GND: pines de tierra.

José E. Martín UPG 2017 ©

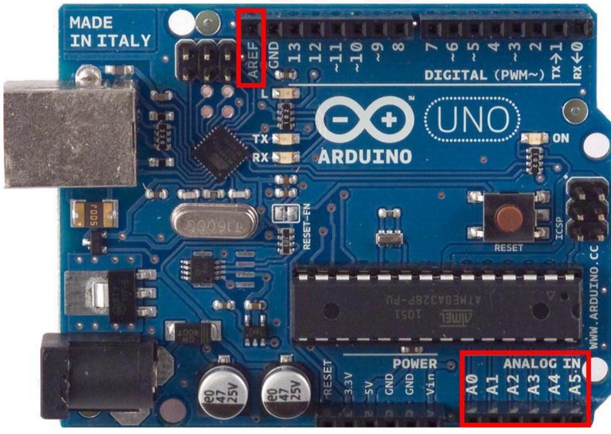
1.4.2 Digital Inputs/Outputs



Cada uno de los 14 pines digitales se puede utilizar como una entrada o salida. Cada pin puede proporcionar o recibir un máximo de 40 mA y tiene una resistencia de pull-up (desconectado por defecto) de 20 a 50 kOhm. Además, algunos pines tienen funciones especializadas como:

- Pin 0 (RX) y 1 (TX). Se utiliza para recibir (RX) y la transmisión (TX) de datos serie TTL.
- Pin 2 y 3. Interrupciones externas. Se trata de pines encargados de interrumpir el programa secuencial establecido por el usuario.
- Pin 3, 5, 6, 9, 10 y 11. PWM (modulación por ancho de pulso).
- Constituyen 8 bits de salida PWM con la función analogWrite ().
- Pin 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). Estos pines son de apoyo a la comunicación SPI.
- Pin 13. LED. Hay un LED conectado al pin digital 13. Cuando el pin es de alto valor, el LED está encendido, cuando el valor está bajo, es apagado.

1.4.3 Analog Inputs



El Arduino posee 6 entradas analógicas, etiquetadas desde la A0 a A5, cada una de las cuales ofrecen 10 bits de resolución (es decir, 1024 estados). Por defecto, tenemos una tensión de 5V, pero podemos cambiar este rango utilizando el pin de AREF y utilizando la función analogReference(), donde le introducimos una señal externa de continua que la utilizara como referencia.

José E. Martín UPG 2017 ©

2. INSTALACIÓN DEL ENTORNO DE TRABAJO ARDUINO.

2.1 DESCARGA E INSTALACIÓN DEL IDE ARDUINO.

Vamos s descargar e instalar el entorno de desarrollo de Arduino (IDE), y comprobar que está correctamente configurado. Para ello vamos a la página de descarga:

http://arduino.cc/en/main/software

Y bajamos la versión más reciente del IDE (Entorno de desarrollo de Arduino)

Elegir la versión correspondiente a nuestro sistema (En Windows recomendamos la versión Installer)

Una vez finalizado, ejecutad el fichero descargado, e ir respondiendo a las opciones de instalación.

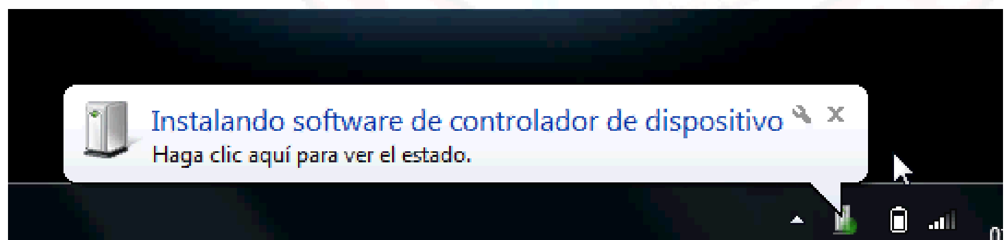
Al cabo de unos minutos finalizara la instalación, y en el área de trabajo del equipo aparecerá el icono de Arduino

2.2 COMPROBACIÓN DE LA INSTALACIÓN.

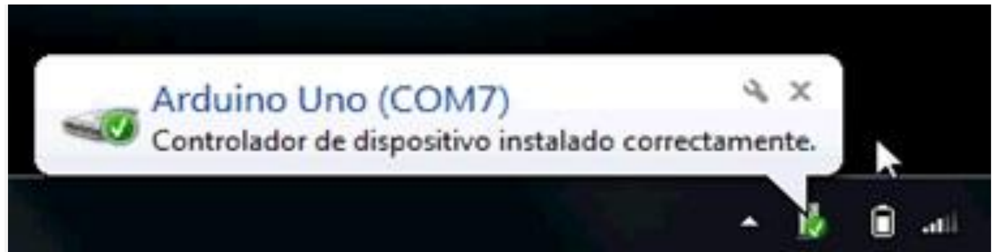
Una vez instalado el IDE, vamos a comprobar que reconoce nuestro Arduino correctamente y que podemos programarlo. Para ello, Conecta tu Arduino a tu ordenador mediante el USB

Comprueba que las luces del Arduino se iluminan indicando que tiene alimentación.

Al hacerlo nuestro PC debe detectar el nuevo dispositivo USB y montar el driver adecuado.

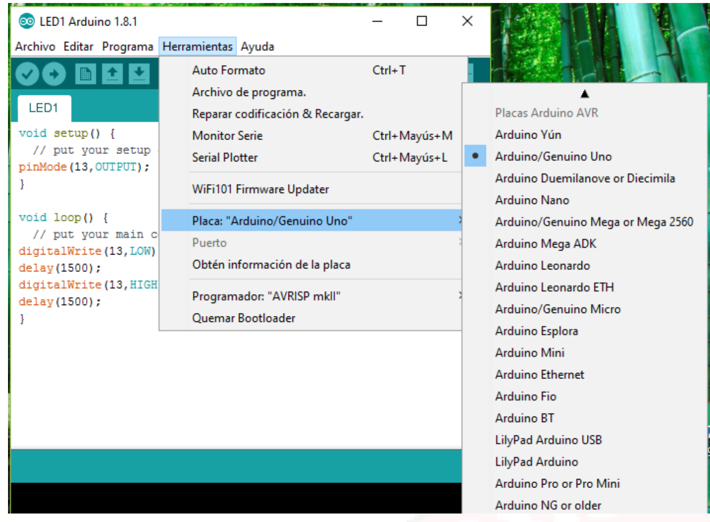


Y finalmente:



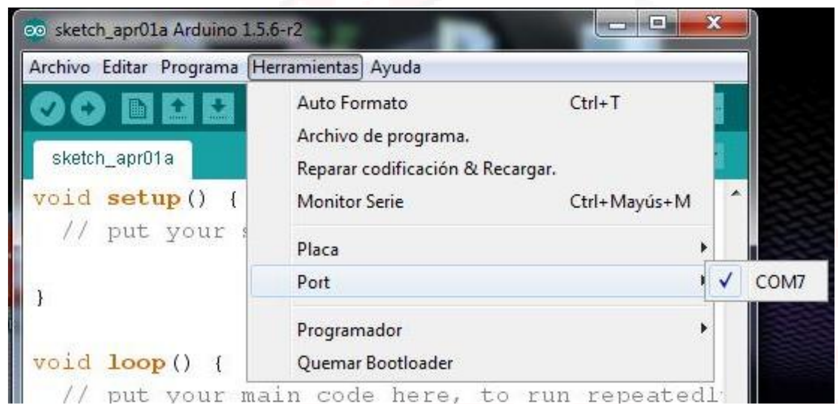
José E. Martín UPG 2017 ©

Atención, el puerto serie en que se instala puede variar del indicado en la imagen, dependiendo de las características del equipo.

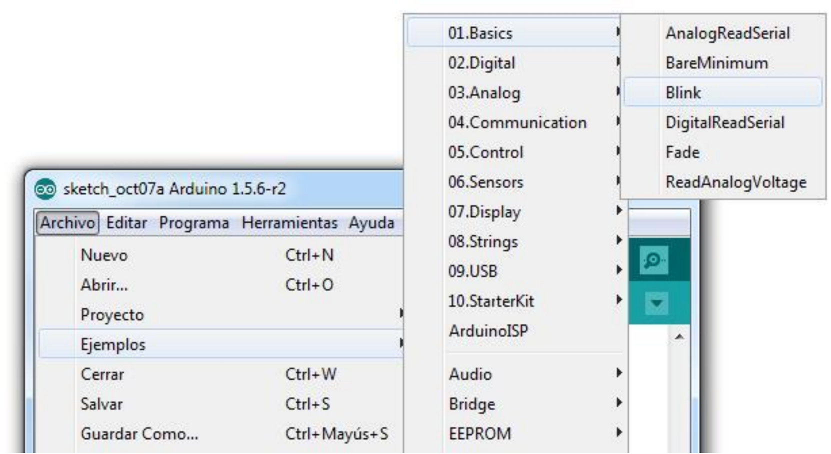


Ahora, ya podemos arrancar el icono de Arduino del escritorio de trabajo y configurar el modelo de Arduino y confirmar el puerto serie al que se conecta. En [Menú]\Herramientas\Placa elegir el modelo exacto de nuestro Arduino. En nuestro caso elegimos un Arduino Uno:

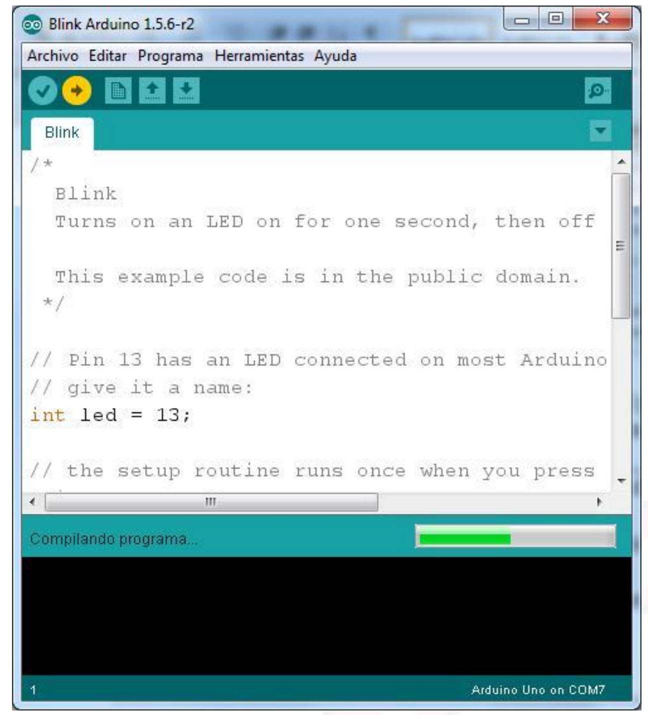
En [Menú]\Herramientas\Port es necesario comprobar que tenemos asignado un puerto y que tiene la marca de selección.



Es importante asignar el puerto y el modelo de Arduino para garantizar el correcto funcionamiento del IDE. La marca de selección debe estar con el tick. Vamos ahora a volcar un programa de ejemplo:



José E. Martín UPG 2017 ©



Al pulsar [Menú]\Archivo\Ejemplos\01.Basics\Blink, aparecerá una serie de textos en el entorno de trabajo, que ignoraremos por ahora. Pulsar el botón Marcado en amarillo, Veremos una línea de progreso verde avanzando.

Si todo va correctamente veremos un mensaje en la parte inferior del IDE:

Este mensaje en color blanco indica que hemos volcado correctamente el programa y ya deberíamos ver una luz que parpadea en nuestra placa Arduino, según indica la flecha roja:

Si vemos la luz parpadeando en nuestro Arduino, enhorabuena, el entorno está instalado y configurado correctamente. Ya podemos pasar a la siguiente sesión.

```
Blink Arduino 1.5.6-r2
Archivo  Editar  Programa  Herramientas  Ayuda
Blink
  Turns on an LED on for one second, then off for one sec

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
Subido
Sketch uses 1.116 bytes (3%) of program storage space.
Maximum is 32.256 bytes.
Global variables use 11 bytes (0%) of dynamic memory,
leaving 2.037 bytes for local variables. Maximum is
2.048 bytes.
13                                      Arduino Uno on COM7
```


CC BY NC José E. Martín UPG 2017 ©

3. PRIMER PROGRAMA

3.1 MATERIAL REQUERIDO.

	Arduino Uno o similar.
	Un cable USB adecuado al conector de tu Arduino.
	Un PC con el entorno de Arduino correctamente instalado y configurado.

3.2 ALGUNAS IDEAS BÁSICAS SOBRE PROGRAMACIÓN.

Un programa de ordenador es básicamente el equivalente a una receta de cocina… pero destinado a un público distinto.

Mientras que las personas somos razonablemente buenas interpretando las instrucciones, generalmente vagas, de una receta de cocina, cuando programamos quien debe entendernos es un ordenador que espera instrucciones precisas respecto a lo que debe hacer y que además carece por completo de la imaginación o capacidad de improvisación humana.

Por ello se desarrollan los lenguajes de ordenador, para dar instrucciones a una máquina de forma:

- Precisa: Sin ambigüedades inherentes a la comunicación humana.
- Univoca: Solo se puede interpretar de una manera.
- Concisa: Preferiblemente ordenes cortas.

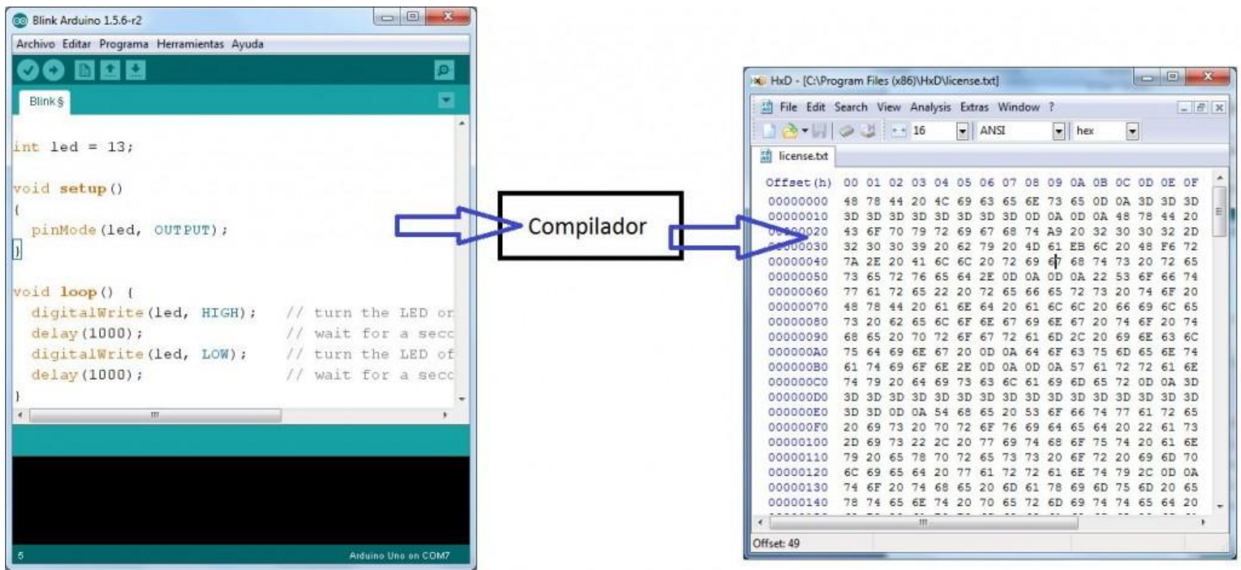
El IDE de Arduino se programa en una variante de C++, que es un lenguaje muy extendido por sus características, aunque no es un lenguaje sencillo. C++, que fija reglas estrictas de cómo escribir estas instrucciones.

Un programa es una serie de instrucciones que se ejecutan en secuencia (salvo que indiquemos expresamente condiciones precisas en las que esta secuencia se altera).

José E. Martín UPG 2017 ©

Un programa interno comprueba que la sintaxis de nuestro programa es acorde a la norma de C++, y si hay cualquier cosa que no le convence dará un error y finalizará la comprobación obligándonos a revisar lo que hemos escrito.

Cuando el comprobador acepta nuestro programa, invoca otro programa que traduce lo que hemos escrito a instrucciones comprensibles para el procesador de nuestro Arduino. A este nuevo programa se le llama compilador.



Esto es lo que nosotros entendemos.

Esto es lo que entiende el procesador.

Funcion del compilador

El compilador convierte nuestras instrucciones (código fuente) en instrucciones del procesador (código ejecutable).

3.3 ESTRUCTURA DE UN PROGRAMA ARDUINO.

Un programa o sketch de Arduino consiste en dos secciones o funciones básicas:

Setup: Sus instrucciones se ejecutan solo una vez, cuando se arranca el programa al encender Arduino o cuando pulsamos el botón de reset. Generalmente incluye definiciones e inicializaciones de ahí su nombre.

Loop: Sus instrucciones se van ejecutando en secuencia hasta el final…. Y cuando acaba, vuelve a empezar desde el principio haciendo un ciclo sin fin.

Cuando abrimos el IDE de Arduino (o hacemos [Menú]\Archivo\nuevo) él nos escribe ya estas dos funciones (en color cobre):

Nótese que el principio de cada función es indicado por la apertura de llave " { " y el fin de la misma corresponde al símbolo de cerrar llaves " } ".

De hecho el conjunto de instrucciones contenidas entre una apertura y cierre de llaves se llama bloque y es de capital importancia a la hora de que nuestro Arduino interprete de una u otra manera las instrucciones que le damos.

José E. Martín UPG 2017 ©

Es imperativo que a cada apertura de una llave corresponda un cierre de llave. En sucesivos capítulos ampliaremos este concepto.

Por ahora resaltar las líneas que aparecen dentro de los bloques principales:

```
// put your setup code here, to run once
// put your main code here, to run repeatedly
```

Cualquier cosa que escribamos precedido por " // " son comentarios, y serán ignorados. Es decir podemos dejarnos mensajes dentro del código,(que de otro modo darían errores). El compilador ignorará cualquier cosa entre // y el fin de línea.

3.4 PRIMERAS INSTRUCCIONES EN ARDUINO C++.

Parece obligado en el mundo Arduino, que el primer programa que hagamos sea el blinking LED, y está bien porque ilustra algunas ideas interesantes en cuanto a sus posibilidades:

La capacidad de Arduino para interactuar con el mundo externo. Algo bastante inusitado para quienes estén acostumbrados a la informática tradicional, donde la potencia de cálculo ha crecido de forma espectacular, pero sigue siendo imposible (o casi), influir en el mundo exterior.

La sencillez del entorno de trabajo. En contraposición a un sistema tradicional de editor / compilador / linker.

Arduino puede relacionarse de diferentes maneras con el mundo que le rodea, Empezaremos por los pines digitales que pueden usarse como:

Entradas: Para leer información digital del mundo exterior.

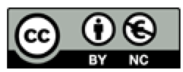
Salidas: Para activar una señal al mundo exterior.

Arduino dispone de 14 pines que pueden ser usados de este modo, numerados del 0 al 13:

pines del 0 al 13

En la sesión anterior cargamos un programa de ejemplo que hacia parpadear un LED en la placa con una cadencia definida. Veamos como programar esto.

Pediremos a Arduino que active su pin 13 como de salida digital y después encenderemos y apagaremos esta señal lo que hará que el LED que tiene conectado de serie se encienda o apague al ritmo que marquemos.


José E. Martín UPG 2017 ©

Para indicar al sistema que deseamos usar el pin 13 como salida digital utilizamos la instrucción:

```
pinMode ( 13, OUTPUT ) ;
```

El primer parámetro indica el pin a usar y "OUTPUT" es para usarlo como salida, y también podría usarse el valor "INPUT" para indicar que vamos a leer de este pin.

Estas definiciones se harán solo una vez al principio, en la función setup(). La nuestra quedará, con una única instrucción que declara que vamos a usar el pin 13 como salida digital:

```
    void setup()
         {
              // initialize the digital pin as an output
              pinMode( 13, OUTPUT) ;
         }
```

Es importante fijarse en que a pesar de ser una única instrucción, hemos delimitado el bloque de esta función mediante abrir y cerrar llaves.

Obsérvese que la instrucción finaliza en " ;" . C++ obliga a acabar las instrucciones con un punto y coma que delimite la orden. Si se omite generará un error.

Para encender el LED usaremos la instrucción:

```
digitalWrite( 13 ,  HIGH) ;
```

Y otra instrucción similar que le ordena apagarlo:

```
digitalWrite( 13 ,  LOW) ;
```

El 13 indica el pin a utilizar y HIGH, LOW indican el valor que deseamos poner en esa salida, que en Arduino corresponden a 5V para HIGH y 0V para LOW.

Si en la función loop() escribiéramos estas dos instrucciones seguidas, Arduino cambiaría estos valores tan deprisa que no percibiríamos cambios, así que necesitamos frenarle un poco para que podamos percibir el cambio.

Para hacer este retraso de, digamos, un segundo, utilizaremos:

```
delay(1000) ;                       //  delay(n) "congela" Arduino n milisegundos
```

Por tanto para programar una luz que se enciende y se apaga, tendríamos que generar una secuencia de órdenes (Como en una receta e cocina) que hicieran:

```
 Informar a Arduino de que vamos a utilizar el pin13 para escribir valores( en el Setup).
Encender el LED : Poner valor alto ( 5V) en dicho pin.
Esperar un segundo.
Apagar el LED: Poner valor bajo (0V) en dicho pin.
Volver a esperar un segundo.
```

Si omitiéramos este segundo retraso, apagaría la luz y volvería a empezar encontrándose la orden de volver a encender. No apreciaríamos que se había apagado.(No espero que me creáis. Comprobadlo).

El procesador de Arduino UNO es muy lento desde el punto de vista electrónico, pero es capaz de conmutar la luz (pasar de encendido a apagado y vuelta a encender) unas 15.000 veces por segundo.

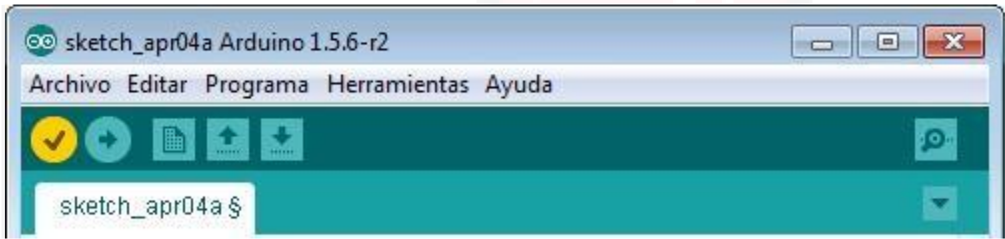
El primer concepto que tenéis que fijar, es que los ordenadores procesan las ordenes en secuencia, una instrucción después de otra y en el orden en que se las dais. *Nuestro programa instruye al ordenador para que ejecute esas instrucciones y fija el orden en el que se ejecutan.*

La forma de escribir un programa en Arduino C++ que haga lo anteriormente descrito es algo parecido a esto:

```
  void setup()
     {
         pinMode( 13 , OUTPUT); // Usaremos el pin 13 como salida
     }
void loop()
     {
        digitalWrite(13 , HIGH); // Enciende el LED
        delay(1000); // Esperar un segundo
        digitalWrite(13 , LOW); // Apagar el LED
        delay(1000); // Esperar otro segundo
    }
```

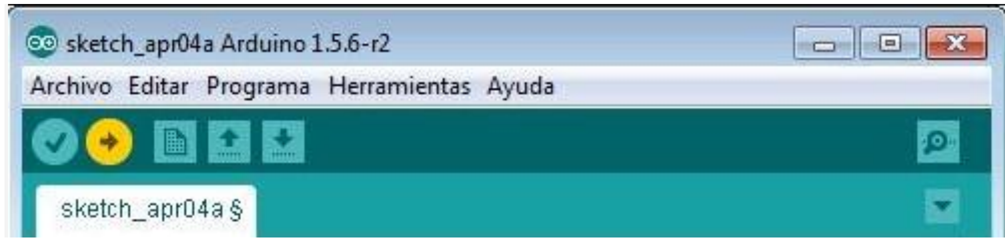
Nótese el sangrado de las líneas para destacar los bloques de código. Esto se considera buena práctica y os lo recomendamos encarecidamente, porque facilita mucho la comprensión del programa.

Solo nos falta ya, comprobar si hay errores y para ello pulsamos el icono en amarillo:



Si todo va bien, (si no hay errores en rojo) podemos compilar y volcar con la siguiente flecha, En caso contrario (y creedme que os pasará con frecuencia) habrá que revisar los posibles errores y corregirlos. Volveremos sobre esto en el futuro.

José E. Martín UPG 2017 ©



La flecha en amarillo volcara nuestro programa al Arduino y podremos comprobar que la luz del pin 13 parpadea con un retraso de un segundo entre encendido y apagado.

Sugerencia: Si modificamos los valores del delay, modificaremos la cadencia del parpadeo.

Nota: Esto no funcionara con ningún otro Pin del Arduino UNO, porque solo el 13 tiene un LED conectado.

José E. Martín UPG 2017 ©

4. PRIMER CIRCUITO.

4.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Un PC con el entorno de Arduino correctamente instalado y configurado.
	Una Protoboard.
	Un diodo LED
330Ω	Una resistencia de 330 Ohmios.
	Algunos cables de Protoboard.

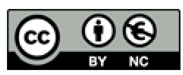
4.2 ALGUNAS IDEAS BÁSICAS SOBRE ELECTRÓNICA

Cuando dejamos fluir agua de un sitio alto a otro más bajo, el agua corre libremente mientras no se lo impidamos, y siempre de arriba abajo. Decimos que las diferentes alturas suponen una diferencia de potencial entre ambos puntos que puede ser transformada en trabajo útil.

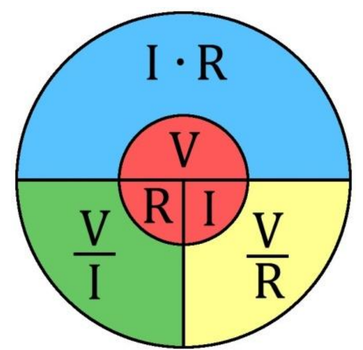
Cuando existe una diferencia de tensión eléctrica (o diferencia de potencial) entre dos puntos con conexión, la electricidad fluye del positivo (o de mas carga) hacia el negativo o menos, y también podemos obtener trabajo útil de este principio.

La idea es que la corriente eléctrica fluye del positivo al negativo porque hay una diferencia de tensión (que medimos en Voltios de símbolo V) pero esto no es una medida absoluta sino la diferencia que hay entre los puntos en que lo medimos.

De la misma manera, la diferencia de altura entre dos puntos solo representa eso, una diferencia y no indica a qué altura se encuentran con respecto a una referencia más o menos arbitraria.


José E. Martín UPG 2017 ©

Hay componentes que se oponen a la libre circulación de la corriente. Los llamamos resistencias, su valor se mide en Ohmios y su símbolo es Ω .



La ley de Ohm, liga todos estos valores de una forma precisa:

V = R x I

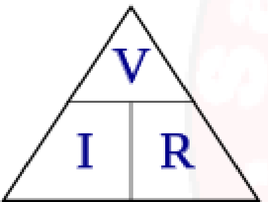
Donde V es la tensión en voltios,

R la resistencia e

I la intensidad eléctrica que fluye.

En un diagrama se muestran las tres formas de relacionar las magnitudes físicas que intervienen en la ley de Ohm V, R e I.

La elección de la fórmula a utilizar dependerá del contexto en el que se aplique. Por ejemplo, si se trata de la curva característica I-V de un dispositivo eléctrico como un calefactor, se escribiría como: I = V/R. Si se trata de calcular la tensión V en bornes de una resistencia R por la que circula una corriente I, la aplicación de la ley sería: V= R I. También es posible calcular la resistencia R que ofrece un conductor que tiene una tensión V entre sus bornes y por el que circula una corriente I, aplicando la fórmula R = V/ I.



triángulo de la ley de Ohm

Una forma mnemotécnica más sencilla de recordar las relaciones entre las magnitudes que intervienen en la ley de Ohm es el llamado "triángulo de la ley de Ohm": para conocer el valor de una de estas magnitudes, se tapa la letra correspondiente en el triángulo y las dos letras que quedan indican su relación (teniendo en cuenta que las que están una al lado de otra se multiplican, y cuando quedan una encima de la otra se dividen como en un operador matemático común).

En el mundo de Arduino la tensión es casi siempre 5V, que es la tensión a que funciona y la que es capaz de poner en sus salidas digitales.

Lo que implica que si la resistencia del circuito es nula (o casi, como en el caso de un cable de cobre) la intensidad de la corriente se dispara y puede llegar a fundir el cable o componente que encuentre.

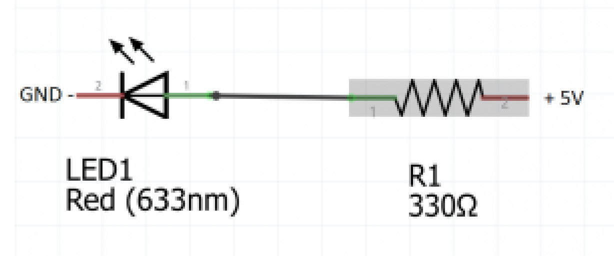
Esto se conoce como cortocircuito o corto simplemente y debe ser evitado decididamente ya que suele acabar con olor a quemado y algún susto, en el mejor caso.

4.3 NUESTRO PRIMER CIRCUITO ELECTRÓNICO

En la sesión anterior programamos el LED conectado al pin 13 de nuestro Arduino. Hoy vamos a duplicar este circuito en el exterior montándolo desde el principio con componentes discretos.

José E. Martín UPG 2017 ©

Su esquema eléctrico sería:



Vemos a la izquierda el símbolo del diodo LED que es emisor de luz y por eso tiene esas flechitas salientes para indicarlo (LED viene del inglés Light Emitting Diode, o diodo emisor de luz).

La resistencia se representa por ese segundo símbolo indicando un nombre R1 y su valor 330Ω.

A su vez vemos a la izquierda las letras GND para indicar que es el negativo. Tiene muchos nombres: Masa, El símbolo –, Tierra(aunque no es lo mismo), Ground, Negativo, cátodo.

Por último a la derecha el símbolo de +5V indica el extremo de tensión positiva o positivo y a veces se representa como Vcc. Las líneas rectas y negras indican conexión eléctrica mediante cables conductores.

Diodo

Un diodo es un componente electrónico de dos terminales que permite la circulación de la corriente eléctrica a través de él en un solo sentido. Este término generalmente se usa para referirse al diodo semiconductor, el más común en la actualidad; consta de una pieza de cristal semiconductor conectada a dos terminales eléctricos. El diodo de vacío (que actualmente ya no se usa, excepto para tecnologías de alta potencia) es un tubo de vacío con dos electrodos: una lámina como ánodo, y un cátodo.

De forma simplificada, la curva característica de un diodo (I-V) consta de dos regiones: por debajo de cierta diferencia de potencial, se comporta como un circuito abierto (no conduce), y por encima de ella como un circuito cerrado con una resistencia eléctrica muy pequeña. Debido a este comportamiento, se les suele denominar rectificadores, ya que son dispositivos capaces de suprimir la parte negativa de cualquier señal, como paso inicial para convertir una corriente alterna en corriente continua. Su principio de funcionamiento está basado en los experimentos de Lee De Forest.

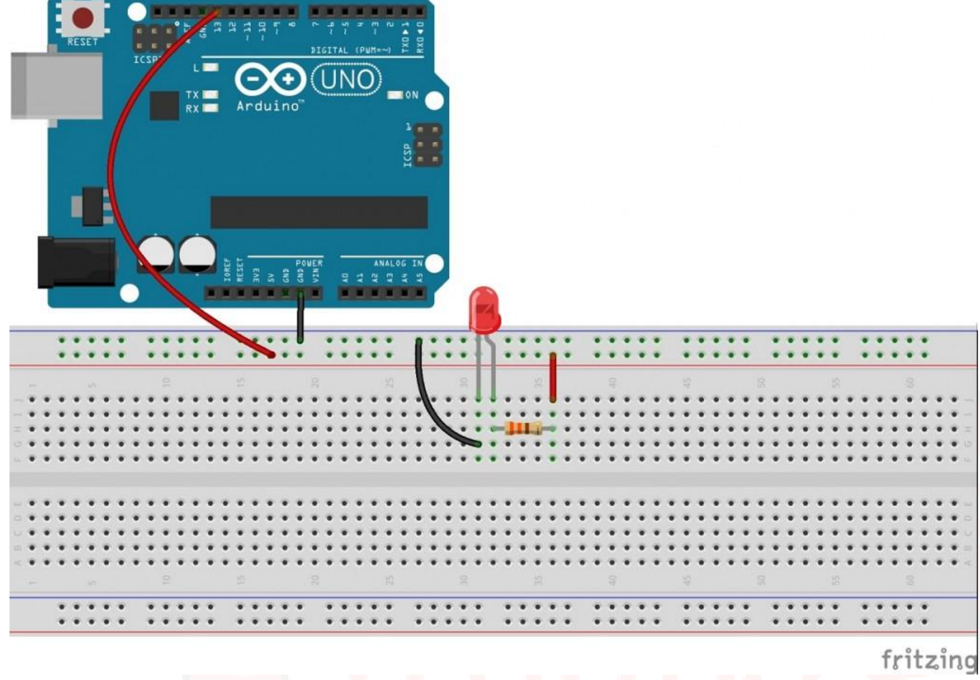
El invento fue desarrollado en 1904 por John Ambrose Fleming, empleado de la empresa Marconi, basándose en observaciones realizadas por Thomas Alva Edison.

Al igual que las lámparas incandescentes, los tubos de vacío tienen un filamento (el cátodo) a través del cual circula la corriente, calentándolo por efecto Joule. El filamento está tratado con óxido de bario, de modo que al calentarse emite electrones al vacío circundante los cuales son conducidos electrostáticamente hacia una placa, curvada por un muelle doble, cargada positivamente (el ánodo), produciéndose así la conducción.

José E. Martín UPG 2017 ©

Evidentemente, si el cátodo no se calienta, no podrá ceder electrones. Por esa razón, los circuitos que utilizaban válvulas de vacío requerían un tiempo para que las válvulas se calentaran antes de poder funcionar y las válvulas se quemaban con mucha facilidad.

Una vez comprendido el esquema eléctrico del circuito, veamos la conexión en la Protoboard:



Este esquema sigue una pauta de marcar los cables que van a positivo en rojo y los que van a GND en negro. Recomendamos encarecidamente se siga esta norma en la práctica porque ayuda a identificar posibles problemas y evita errores.

La Protoboard une los puntos de la línea azul entre si y los de encima de la línea roja entre sí, (se les llama raíles), pero no conecta el raíl rojo positivo con el raíl negro negativo.

A su vez existen dos zonas de líneas verticales en la Protoboard. Estas líneas verticales están unidas entre sí internamente, para facilitar la conexión de los componentes, pero no se unen las líneas paralelas.

Las claves para montar el circuito con éxito, son:

- Conectamos el pin 13 de Arduino a la línea roja de la Protoboard: Positivo.
- Conectamos el GND de Arduino a la línea azul de la Protoboard: Ground.
- Usamos el raíl positivo (los pines de la línea roja) para conectar a la resistencia.
- El otro extremo de la resistencia se conecta al positivo del LED porque están en la misma vertical de la Protoboard (y esta los conecta eléctricamente).
- Nótese que el positivo del LED está claramente marcado como de mayor longitud mediante un pequeño ángulo cerca de la base.

José E. Martín UPG 2017 ©

- Un diodo LED casi no presenta resistencia propia, por lo que siempre debe usarse una resistencia adicional que limite el paso de corriente, y evite que se queme. (Una resistencia entre 220 y 3300 Ω suele ser adecuada).
- El circuito se cierra con un cable desde el negativo del LED al raíl de GND.
- Cuando nuestro programa ponga un valor de HIGH (5V) en el pin 1,3 permitirá el flujo de corriente por el circuito iluminando el LED. Con LOW sencillamente el circuito estará apagado, sin tensión.

Podemos ahora volcar el programa que hicimos en la sesión 2 (o simplemente cargar el ejemplo Blink), siguiendo el procedimiento que definimos allí, y veremos cómo ésta vez, además del LED propio de Arduino, nuestro LED exterior parpadea siguiendo el mismo ciclo de encendido y apagado.

José E. Martín UPG 2017 ©

5. CIRCUITO CON MÚLTIPLES LEDS.

5.1 MATERIAL REQUERIDO.

	Arduino Uno o similar Un PC con el entorno de Arduino correctamente instalado y configurado.
	Una Protoboard.
	8 x diodos LED.
330Ω	Una resistencia de 330 Ohmios.
	Algunos cables de Protoboard.

5.2 UN CIRCUITO CON VARIOS LED

Si quisiéramos montar un circuito que tuviera 8 LEDs y en el que la luz se desplazara de uno a otro, una posibilidad sería repetir varias veces las mismas secuencias de instrucciones que ya conocemos.

Por ejemplo si conectamos distintos LEDs a distintos pines digitales de Arduino, deberíamos declararlo en nuestra Función de setup() que podría ser:

```
void setup()
   {
          // initialize the digital pins as an output
         pinMode( 13, OUTPUT) ;
         pinMode( 12, OUTPUT) ;
         pinMode( 11, OUTPUT) ;
         …………………………
         pinMode( 6, OUTPUT) ;
```


José E. Martín UPG 2017 ©

```
}
```

Y a su vez nuestro loop() debería repetir tantas veces como LEDs tengamos el juego de encender y apagar cada uno de los LEDs en secuencia desde el pin 13 hasta el 6.

Esta solución es la que podríamos describir como de fuerza bruta, pero no es muy elegante, es trabajosa y probablemente cometeríamos más de un error al escribirla, porque las personas tendemos a equivocarnos haciendo tareas repetitivas aburridas (y esta lo es mortalmente, imaginad un circuito de de 16 LEDs).

En cambio los ordenadores no se aburren y además C++ nos ofrece un medio cómodo de indicarle que debe repetir algo un número definido de veces. Este medio es la instrucción For que podemos usar en combinación con una variable.

Una variable es un contenedor que puede tomar varios valores, en nuestro caso aceptará todos los valores entre 6 y 13.

C++ nos exige declarar el tipo de las variables antes de usarlas. En nuestro caso usaremos el tipo entero que se escribe int para indicar que esta variables es numérica y entera, sin decimales.

Iremos viendo que existen otros tipos de variables. Volveremos sobre este tema en próximas sesiones.

Así por ejemplo, para inicializar en nuestro setup() los pines desde el 13 hasta el 6 como salidas (requerido por nuestro Arduino) podríamos usar la instrucción for de la siguiente manera:

```
void setup()
    {
        int i = 0 ; // Inicializamos la variable i como un entero
        for ( i = 6 ; i < 14 ; i++)
              pinMode( i , OUTPUT) ;
    }
```

Aunque la sintaxis parece complicada al principio, uno se acostumbra con rapidez. Aquí lo importante es que for necesita 3 parámetros separados por un carácter de punto y coma.

Estos parámetros son y en éste orden:

Una variable que ira tomando valores según una cierta regla, y a la que asignamos un valor inicial. En este caso: i = 6.

El ciclo continúa mientras se cumpla esta condición. En nuestro caso mientras la i sea menor que 14, o sea hasta el 13: i <14

Como cambia la variable en cada iteración. En nuestro caso i++ que es pedirle a C++ que incremente en uno la variable i, al final de cada iteración.

Con el mismo criterio podríamos escribir la función loop() así:

```
    void loop()
        {
```

José E. Martín UPG 2017 ©

```
            int i = 0 ; // Inicializamos la variable i como un entero
            for ( i = 6 ; i < 14 ; i++)
                  {
                     digitalWrite( i , HIGH) ;
                     delay (500) ;
                     digitalWrite( i , LOW);
                     delay (500) ;
                  }
        }
```

En la sesión 3 el código era muy similar excepto en que escribíamos el valor 13 para el único pin que tenía un LED conectado. Aquí asignamos el pin con una variable i , que va tomando los valores de 6 hasta el 13 para el pin.

Nótese que la instrucción for no lleva un punto y coma al final. Esto es porque se aplica al bloque de instrucciones que le siguen entre llaves, como es el caso del loop() La iteración realiza las cuatro instrucciones que siguen a la línea del for, porque están dentro de un bloque de instrucciones.

Las instrucciones que se aplican a bloques de código, no llevan punto y coma al final.

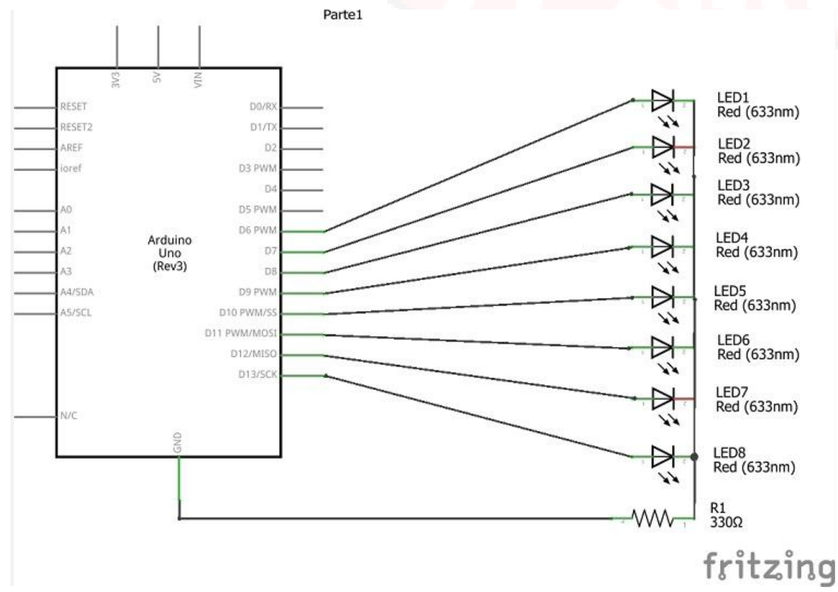
En el caso de particular de que el bloque lleve una única línea de código, las llaves pueden ser omitidas, como en el caso de la instrucción for en la función setup() de arriba.

5.3 ESQUEMA ELECTRÓNICO DEL CIRCUITO

El esquema del circuito es muy similar al presentado anteriormente, salvo por el hecho de que colocamos en la Protoboard 8 LEDs.

La única novedad es que dado que la función de la resistencia es limitar la intensidad de la corriente que circula por el circuito, y puesto que todos los diodos tienen masa común, basta una única resistencia entre este punto y Ground.

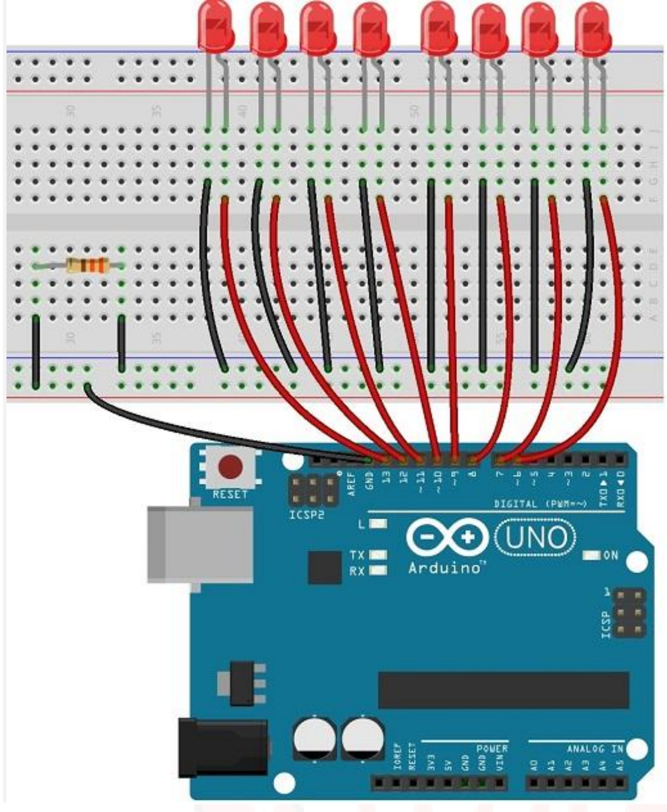
Cuando nuestro programa levante el pin correspondiente a valor a HIGH, se cerrar el circuito iluminándose el LED asociado.



José E. Martín UPG 2017 ©

Con este circuito, y con el programa 4.1 descrito en las páginas anteriores, tendremos un efecto de luces similar al del "coche fantástico" (O de los Zylon para los aficionados a la ciencia ficción).

A continuación incluimos un esquema de conexión del circuito en una protoboard.



En general, se considera buena costumbre (la recomendamos), montar los circuitos que veamos a partir del esquema electrónico del mismo, más que a partir del diagrama de conexiones de la Protoboard.

La razón es que con el esquema, la comprensión del circuito es completa y se evita la tentación de copiar la práctica sin necesidad de entenderla.

Además, el diagrama electrónico del circuito es su completa descripción y suele resultar más sencillo comprender la función del mismo. En cambio a medida que los circuitos se hacen más complejos, comprender su función desde un esquema de Protoboard puede complicarse mucho, y peor aún llevar a una interpretación errónea.

5.4 VARIANTES DEL PROGRAMA CON EL MISMO CIRCUITO

Este montaje nos permite jugar con las luces y se presta a varios programas diferentes para conseguir distintos efectos.

Por ejemplo, con el programa anterior 4.1, el efecto no es exactamente el del coche fantástico porque cuando acabamos de iterar el for, el programa vuelve a empezar desde el principio, lo que hace que la luz salte desde el pin 6 hasta la del pin 13.

Así pues ¿Podríamos hacer que la luz rebotara? Pensadlo un poco.

Desde luego que sí, bastaría con usar dos ciclos for, similar a lo siguiente:

José E. Martín UPG 2017 ©

```
void loop() // Prog_4_2
  {
     for ( int i = 6 ; i < 14 ; i++) // Definimos la variable i sobre la marcha
             {
                    digitalWrite( i , HIGH) ;
                    delay (500) ;
                    digitalWrite( i , LOW);
                    delay (500) ;
              }
     for ( int i = 12 ; i >6 ; i--) // Definimos la variable i sobre la marcha
             {
                   digitalWrite( i , HIGH) ;
                   delay (500) ;
                   digitalWrite( i , LOW);
                   delay (500) ;
             }
  }
```

El primer ciclo for hace que las luces se encienda en secuencia desde la 6 hasta la 13. El segundo bucle entra a continuación empezando con la luz 12 (para no repetir la 13) y finalizando con la 7(para no repetir la 6), y vuelta a empezar.

En el segundo bucle hemos hecho una cuenta atrás diciéndole a la variable i que se decrementara en uno en cada iteración mediante la instrucción i– .

También nos hemos aprovechado de que C++ nos permite definir variables sobre la marcha dentro de la propia instrucción for, sin necesidad de dedicarle una línea completa a la declaración e inicialización.

Otra variante seria, hacer un efecto de ola en al que las luces subieran dejando encendidos los LEDs previos hasta alcanzar el máximo y ahora descender apagando los LEDs superiores. Os recomendamos intentar resolver el problema como desafío, antes de buscar una solución.

Programar es en parte aprender las instrucciones de un lenguaje (la parte fácil), y otra más difícil que es aprender a resolver los problemas de un modo que nos permita darle instrucciones a un ordenador para que lo lleve a cabo.

Estos procedimientos secuenciales de cómo resolver un cierto tipo de problemas es lo que se conoce como un algoritmo.

Según el problema que abordemos el algoritmo será más o menos complicado pero aprender a programar tiene más que ver con desarrollar esta capacidad de resolver problemas lógicos en una secuencia de pasos que podamos codificar en un ordenador.

José E. Martín UPG 2017 ©

Por cierto, cualquiera puede aprender a programar. No lo dudéis. Solo que como en todo, a unos les lleva más tiempo que a otros desarrollar la habilidad necesaria. Al principio muchos me dicen que les duele la cabeza de pensar en este tipo de cosas, pero os animo a continuar (poco a poco si es preciso) porque os encontrareis que vale la pena.

José E. Martín UPG 2017 ©

6. LAS ENTRADAS DIGITALES DE ARDUINO.

6.1 MATERIAL REQUERIDO.

	Arduino Uno o similar.
	Una Protoboard.
	Un diodo LED.
	Un pulsador.
330Ω	Dos resistencias de 330 Ohmios.
	Algunos cables de Protoboard.

6.2 ENTRADAS DIGITALES

Con frecuencia en electrónica necesitamos saber si una luz está encendida o apagada, si alguien ha pulsado un botón o si una puerta ha quedado abierta o está cerrada.

A este tipo de señales todo / nada, SI / NO, TRUE /FALSE, 0/1 se les llama digitales, y podemos manejarlas con los pines de 0 al 13 de Arduino y por eso hablamos de pines digitales.

Muchos de los sensores y actuadores que vemos en el mundo real son digitales:

Como actuadores digitales, tenemos luces, alarmas, sirenas, desbloqueo de puertas, etc.

Como sensores digitales podemos mencionar botones y pulsadores, Finales de carrera, desbordamiento de nivel, sensores de llamas, humo o gases tóxicos.

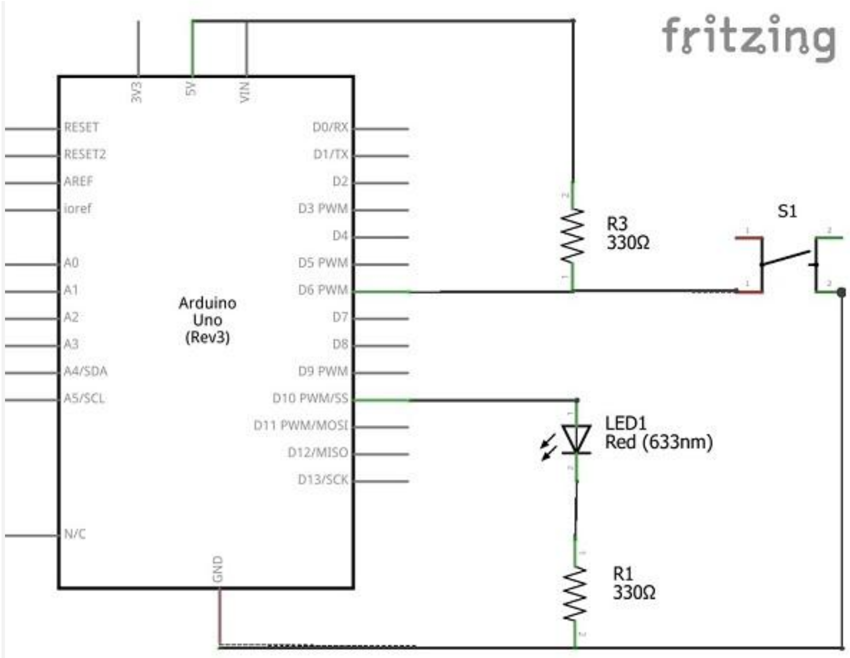
Hemos visto que Arduino pueden usar los pines digitales como salidas todo o nada para encender un LED. De la misma manera podemos leer valores, todo o nada, del mundo exterior.

José E. Martín UPG 2017 ©

En esta sesión veremos que los pines digitales de Arduino pueden ser usados tanto de entrada como de salida. Vamos a leer un botón o pulsador externo y vamos a encender o apagar un LED en función de que el botón se pulse o no.

6.3 ESQUEMA ELECTRÓNICO DEL CIRCUITO.

Montaremos un circuito con un diodo LED y resistencia conectado al pin digital 10 de Arduino, tal y como vimos en las sesiones previas y además un segundo circuito con un pulsador S1 conectado al pin 6 con una resistencia como se muestra en el diagrama siguiente.



Obsérvese que mientras no pulsemos S1 el pin 6 de Arduino está conectado a 5V a través de la resistencia R3 forzando una lectura de tensión alta (HIGH). En cambio cuando pulsemos S1 cerraremos el circuito del pin 6 a Ground con lo que leerá tensión baja, LOW. En ambos casos tenemos un valor de tensión definido.

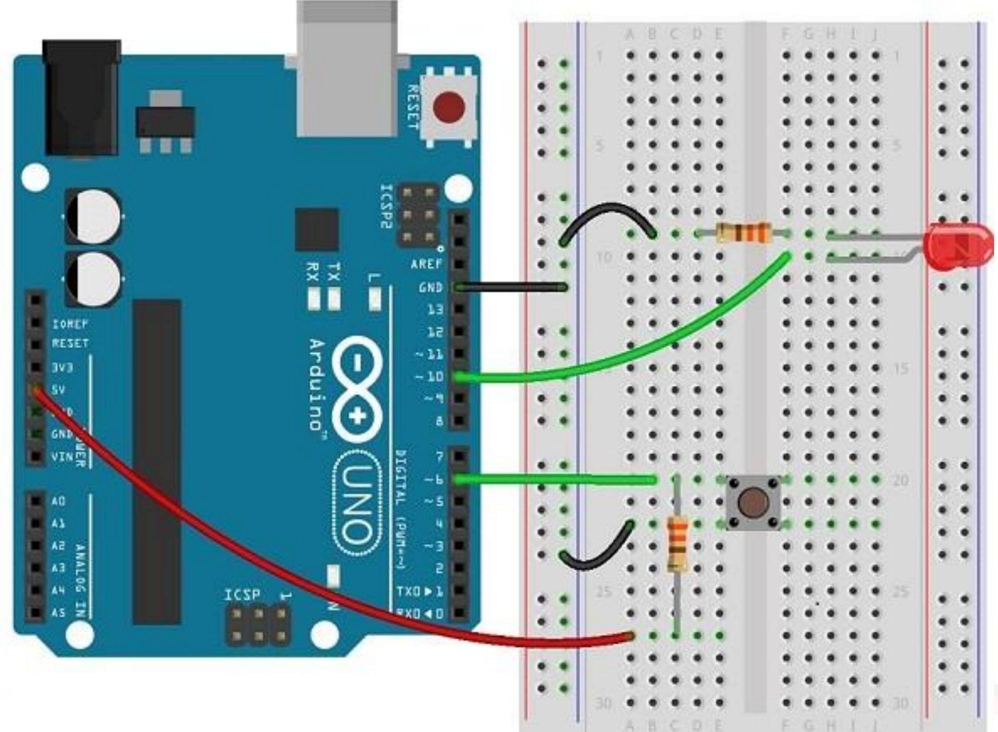
Si no pusiéramos la resistencia R3, al pulsar S1 leeríamos correctamente LOW en el pin 6. Pero al dejar de pulsar S1 el pin 6 estaría en un estado flotante, que es ni HIGH ni LOW sino indeterminado. Como esto es inaceptable en circuitos digitales forzamos una lectura alta con R3.

A esta resistencia que fuerza el valor alto en vacio se le conoce como pullup Si la conectáramos a masa para forzar una lectura a Ground se le llamaría pulldown resistor.

Esta resistencia es clave para que las lecturas del pulsador sean consistentes. El circuito, simplemente, no funcionará bien si se omite (volveremos sobre esto).

Y aquí tenemos el esquema para protoboard del circuito.

José E. Martín UPG 2017 ©



En este esquema hemos seguido la práctica habitual de usar cables negros para conectar a masa y cables rojos para conectar a tensión (5V).

Obsérvese que el pulsador S1 tiene cuatro pines (el que está sobre la resistencia horizontal). Esto es porque cada entrada del interruptor tiene dos pines conectados. En nuestro circuito simplemente ignoramos los pines secundarios.

6.4 LEYENDO LOS PULSADORES

Empecemos haciendo un programa que haga que el LED se encienda cuando pulsamos el botón y se apague cuando lo soltamos. Para ello pediremos a Arduino que configure el pin digital 10 (D10) como salida para manejar el LED, y el pin digital 6 (D6) como entrada para leer el botón.

Normalmente en programas sencillos basta con poner el número de pin en las instrucciones. Pero a medida que el programa se complica esto tiende a provocar errores difíciles de detectar.

Por eso es costumbre definir variables con los números de pin que usamos, de forma que podamos modificarlos tocando en un solo lugar (y no teniendo que buscar a lo largo del programa). Vamos a escribir esto un poco más elegantemente:

```
int LED = 10 ;
int boton = 6;
 void setup()
     {
          pinMode( LED, OUTPUT) ; // LED como salida
          pinMode( boton   , INPUT) ;             //botón  como entrada
     }
```


BY NC
José E. Martín UPG 2017 ©

Atención: C++ diferencia entre mayúsculas y minúsculas y por tanto LED, Led y led no son lo mismo en absoluto. Del mismo modo, pinMode es correcto y en cambio pinmode generará un error de compilador fulminante.

He usado la variable botón sin acento porque no es recomendable usarlos ni la ñ en los nombres de variables, porque pueden pasar cosas extrañas.

Vimos que para encender el LED bastaba usar digitalWrite(LED, HIGH). Para leer un botón se puede hacer algo similar: digitalRead(botón). Veamos cómo podría ser nuestro loop:

```
void loop()
 {
       int valor = digitalRead(boton) ; //leemos el valor de boton en valor
       digitalWrite( LED, valor) ;
    }
```

¿Fácil no? Aunque el LED está encendido hasta que pulsamos el botón y se apaga al pulsar.

¿Cómo podríamos hacer lo contrario, que el LED se encienda al pulsar y se apague si no? Bastaría con escribir en LED lo contrario de lo que leamos en el botón.

Existe un operador que hace eso exactamente el operador negación " ! " . Si una valor dado x es HIGH, entonces !x es LOW y viceversa.

Un operador es un símbolo que relaciona varios valores entre sí, o que modifica el valor de una variable de un modo previsible.

Ejemplos de operadores en C++ son los matemáticos como +,-,* , / ; y hay otros como la negación ! o el cambio de signo de una variable : – x. Iremos viendo más.

De hecho este tipo de operaciones son tan frecuentes que C++ incorpora un tipo llamado bool o booleano que solo acepta dos valores TRUE (cierto) y FALSE y son completamente equivalentes al 1 / 0, y al HIGH / LOW

Este nuevo programa sería algo así:

```
void loop()
    {
      int valor = digitalRead(boton) ;       // leemos el valor de boton en valor
      digitalWrite( LED, !valor) ;           //Escribimos valor en LED
    }
```

Hemos definido valor como bool, porque podemos usar el valor de tensión alto como TRUE y el valor bajo como FALSE.

SI el botón no está pulsado el D6 leerá TRUE y por tanto pondrá LED a FALSE. En caso contrario encenderá el LED.

De hecho podríamos escribir una variante curiosa del blinking LED usando el operador negación:

José E. Martín UPG 2017 ©

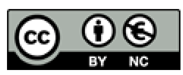
```
void loop()
    {
        bool valor = digitalRead (LED) ;
        digitalWrite(  LED, !valor) ;
        delay ( 1000) ;
    }
```

Podemos leer la situación actual de un pin (nos devuelve su estado actual), aún cuando lo hayamos definido como salida, En cambio no podemos escribir en un pin definido como entrada.

La primera linea lee la situación del LED y la invierte en la segunda línea, después escribe esto en LED. Y puestos a batir algún record, podemos escribir el blinking led en solo dos líneas:

```
void loop()
     {
          digitalWrite(  LED , ! digitalRead( LED)) ;
          delay ( 1000) ;
     }
```

Las instrucciones dentro de los paréntesis se ejecutan antes que las que están fuera de ellos. Por eso el digitalRead se ejecuta antes que el digitaWrite..


José E. Martín UPG 2017 ©

7. CONDICIONALES Y BOTONES.

7.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Un PC con el entorno de Arduino correctamente instalado y configurado.
	Una Protoboard.
	Un diodo LED.
	Un pulsador.
330Ω	Dos resistencias de 330 Ohmio.
	Algunos cables de Protoboard..

7.2 LÓGICA DIGITAL Y ALGEBRA DE BOOL

En la sesión anterior presentamos el tipo de variable bool destacando que solo puede tomar dos valores: True o False. Aunque para quienes no estén acostumbrados al algebra booleana o binaria puede parecer excesivo dedicar un tipo a algo tan simple, en la práctica buena parte de las instrucciones de programación se apoyan o dependen de este tipo de variables.

La razón práctica es que con frecuencia hay que tomar decisiones para seguir un camino u otro en función de que se cumpla una condición dada; Esta condición se debe evaluar necesariamente, a True o False para tomar una decisión sin duda posible.

Por ejemplo, en las sesión 4 usamos la instrucción for y comentamos que la iteración se mantiene mientras se cumpla una cierta condición. Esta condición debe ser evaluable a True o False, es decir es un booleano.

José E. Martín UPG 2017 ©

Existen otras muchas instrucciones que se apoyan en los valores booleanos, (como los condicionales if que veremos en esta sesión) pero en un modo muy explicito toda la computación actual se basa en la lógica digital de solo dos valores que solemos llamar 1 y 0, pero que con todo derecho podemos llamar a estos valores True y False.

Los ordenadores modernos funcionan mediante la aplicación del algebra de bool a variables booleanas y con un juego completo de operadores lógicos como la negación, que vimos en la sesión anterior, mas operadores lógicos como AND, OR, + y – .

7.3 LA INSTRUCCIÓN IF

En este capítulo vamos a presentar unas instrucciones nuevas de C++, que nos permitan tomar decisiones para hacer una cosa u otra.

La instrucción if es muy sencilla de usar, basta con pasarle entre paréntesis una variable o condición que se evalúe a true o false. Si el resultado es true se hace el bloque que viene a continuación y en caso contrario se ejecuta el bloque que hay detrás del else si existe.

Si no existe la clausula del esle, entonces el bloque que sigue al if, se ejecuta o no, en función de la condición y luego sigue con la secuencia de instrucciones a continuación.

```
    if ( condición)
         {
             instrucción 1 ;
             instrucción 2 ;
             ................
         }
    else
        {
             instruccion20 ;
             instruccion21 ;
             ..............
}
```

Recordemos que en el circuito de la sesión anterior disponíamos de un pulsador y de un LED, en esta sesión vamos a continuar con el mismo circuito y para conseguir que el LED se encienda o apague al pulsar el botón. Para ello podríamos mantener la misma función setup() y escribir el loop() diferente:

```
    void loop()
         {
             bool valor = digitalRead(boton) ;
             if ( valor)
                  digitalWrite( LED, HIGH) ;
             else
                  digitalWrite( LED, LOW) ;
          }
```

José E. Martín UPG 2017 ©

Leemos primero el botón a una variable bool y después decidimos si encender o apagar el LED dependiendo de qué su valor sea True o False.

Recordemos que un bloque es un conjunto de instrucciones encerrados entre llaves y que hay un caso particular en el que se pueden omitir si y solo si, el bloque consta de una única instrucción como es nuestro caso.

Se puede utilizar una instrucción if omitiendo la cláusula de else si se desea, pues esta es opcional.

7.4 VOLVIENDO CON LOS BOTONES

Vamos con un programa diferente. Queremos que el botón actúe como un interruptor, que al pulsarlo una vez se encienda, y la próxima vez lo apague. Podríamos plantear algo así y os recomiendo que lo probéis en vuestro Arduino:

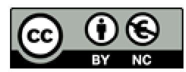
```
int LED = 10 ; int boton = 6 ;
bool estado = false ;
void setup()
     {
        pinMode( LED, OUTPUT) ;
        pinMode( boton , INPUT_PULLUP) ;
        digitalWrite(LED , LOW) ; // Apagamos el LED al empezar
     }
void loop()
     {
         bool  valor =  digitalRead(boton) ; //leemos el botón: false  =  LOW
         if ( valor == false )               // esto es que han pulsado el botón
              {
                   estado = ! estado ;           // cambiamos el estado
                   digitalWrite(LED, estado) ;  // escribimos el nuevo valor
              }
     }
```

La idea es definir una variable llamada estado al principio para guardar la situación del LED. El loop comprueba si se ha pulsado el botón, y de ser así invierte su estado, y después escribe el valor de estado en el LED. Si estaba encendido lo apaga. Si estaba apagado se enciende.

Aunque parece un plan perfecto, en la práctica no va a funcionar. En el tiempo que nosotros tardamos entre pulsar y liberar el botón, nuestro humilde Arduino es capaz de leer unos cuantos miles de veces el pulsador e invertir el valor del LED otras tantas.

Por eso, si lee un número par de veces dejara el LED como estaba y si lo lee un número impar de veces lo invertirá. En la práctica la situación del LED se torna aleatoria, y si pulsáis repetidamente el botón veréis que el resultado es impredecible.

Otra fuente de problemas es que en el mundo real un interruptor no cambia de un estado a otro de forma perfecta, sino que suele rebotar y causar varios conexiones y



José E. Martín UPG 2017 ©

desconexiones muy rápidas antes de quedar en un valor estable. A esto se le llaman rebotes (bouncing) y al procedimiento para eliminar estos rebotes se le llama debouncing en la jerga electrónica.

El debouncing se puede hacer por hardware con un conjunto de resistencia y condensador, o por software, mucho más frecuentemente (por más barato) y para esto una solución es nuevamente frenar a Arduino y hacerle esperar un tiempo entre 50 y 250 mili-segundos una vez que detecta que se ha pulsado el botón, de modo que nos dé tiempo a liberar el pulsador:

```
void loop()
      {
          bool  valor =  digitalRead(boton) ; //leemos el botón: false = LOW
         if ( valor == false )              // esto es que han pulsado el botón
             {
                 estado = ! estado ;         // cambiamos el estado
                 digitalWrite(LED, estado) ; // escribimos el nuevo valor
                 delay(250) ;
             }
       }
```

Muy importante: Nótese que la condición es (valor == false), con doble = . En C++ la comparación de dos valores usa ==, la asignación de valor a una variable solo uno. Esto es fuente de errores frecuentes al principio (entre novicios inexpertos).

Este lapso de 250 ms es suficiente para pulsar y liberar el botón cómodamente. Si probáis esta variante veréis que ahora el LED invierte su valor cada vez que pulsas, siempre y cuando no te demores demasiado en liberar el botón.

Pero… ¿Qué pasa cuando dejas el botón pulsado?

Pues sencillamente que el LED invierte su estado cada 250 ms (milisegundos) y tenemos otra variante del blinking LED.

Si queremos poder mantener pulsado sin que se produzca este efecto hay que sofisticar un poco más el programa:

```
int LED = 10 ; int boton = 6 ;
bool estado = true ;
bool estado_anterior = true ;

void setup()
     {
          pinMode(boton, INPUT_PULLUP);     //Hemos eliminado R3
          pinMode(LED, OUTPUT);
     }

void loop()
```

José E. Martín UPG 2017 ©

```
         {
               estado = digitalRead(boton);
               if (estado != estado_anterior)   //hay cambio : Han pulsado o soltado
                  {
                  if (estado == LOW) //Al pulsar botón cambiar LED, pero no al soltar
                     digitalWrite(LED, !digitalRead(LED));
                     estado_anterior = estado ;  // Para recordar el ultimo valor
                  }
          }
```

Ya dijimos que para comprobar si dos valores son iguales usamos ==, Para comprobar si son diferentes usamos != , y existen otros operadores relacionales

```
Igual que: ==
Distinto de: !=
Mayor que: >
Mayor o igual: >=
Menor que: <
Menor o igual: <=
```

Vale la pena comentar aquí que, a pesar de su aparente inocencia, los botones tienen una sorprendente habilidad para complicarnos la vida, y que en la práctica la combinación de rebotes y la necesidad de corregirlos, junto al uso de pullups que garanticen la correcta lectura, pueden hacer que su uso se pueda complicar mucho más de lo que parece, sino se estudia el problema con calma.

Por último, una condición lógica se puede construir mediante los operadores lógicos AND, OR, y NOT cuyos símbolos son respectivamente: &&, | | y !

Si usáramos un circuito dos pulsadores con pullups (True, si no se pulsa) y un LED, dependiendo del comportamiento que se busque podemos especificar diferentes condiciones:

```
If ( boton1 && boton2) Que ambos botones estén sin pulsar
If ( !( boton1 && boton2)) Que ambos estén pulsados.
If( boton1 || boton2 ) Que al menos uno este sin pulsar, o ambos.
```


José E. Martín UPG 2017 ©

8. COMUNICACIÓN CON EL EXTERIOR.

8.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Un PC con el entorno de Arduino correctamente instalado y configurado.

8.2 COMUNICACIÓN SERIE CON EL MUNDO EXTERIOR

Más antes que después, vamos a necesitar comunicar nuestro Arduino con nuestro PC. Las razones son varias, enviarle órdenes o recibir información o señales por ejemplo.

Los PCs disponen de teclados, pantallas y adaptadores de red, pero con Arduino tenemos que usar el puerto USB que establecerá una conexión en serie con nuestro PC.

La comunicación en serie es muy sencilla, bastan dos hilos para enviar una diferencia de tensión entre ellos y poder marcar niveles alto (5V) y bajo(0V) y con esto podemos transmitir información digital. Ahora solo nos falta pactar dos cosas entre quien envía y quien recibe:

Un código común para codificar los caracteres que enviamos.

Un acuerdo de velocidad para saber a qué ritmo hay que leer los datos.

El código común que vamos a usar con Arduino se llama código ASCII y es estándar en todos los PCs. Es una forma de codificar las letras mediantes números que representas estos caracteres. Recordad que solo podemos transmitir unos y ceros.

Así por ejemplo la letra A se representa por el numero 65, la B el 66, C el 67… Prácticamente todos los PCs actuales utilizan este código y eso incluye a Windows, Mac y Linux (y por eso podemos leer emails enviados desde distintas plataformas), pero es importante comprender que esté es uno más entre varios códigos de caracteres posibles (EBCDIC por ejemplo).

Actualmente, en realidad, se suele usar una extensión del código ASCII (llamada Unicode) que permita el uso de caracteres no incluidos en la tabla original, y que permita representar caracteres como las Ñ, o acentos para el español, pero también alfabetos distintos como el Kanji chino o el alfabeto cirílico. Y este es el motivo por el que podéis leer las letras chinas o rusas en las páginas de internet de estos países.

El otro factor a pactar para realizar una comunicación serie es la velocidad. Dado que solo disponemos de dos hilos para transmitir, necesitamos saber cuándo hay que leer la línea y esto se hace estableciendo un acuerdo de velocidad. Si la velocidad de envío es distinta de la velocidad de lectura, el mensaje final será irreconocible.

José E. Martín UPG 2017 ©

Buena parte de los errores de comunicación serie programando con Arduino se suelen deber a una diferencia de velocidad entre el emisor y el receptor.

Esta velocidad se mide en bits por segundo y vamos a ver que Arduino soporta diferentes velocidades de comunicación serie.

8.3 ESTABLECIENDO LA COMUNICACIÓN SERIE

Arduino dispone de una librería serie incluida llamada Serial, que nos permite envía información al PC y para usarla simplemente tenemos que pedirle en nuestro setup() que la incluya. La instrucción que se encarga es:

```
Serial.begin( velocidad )  ;
```

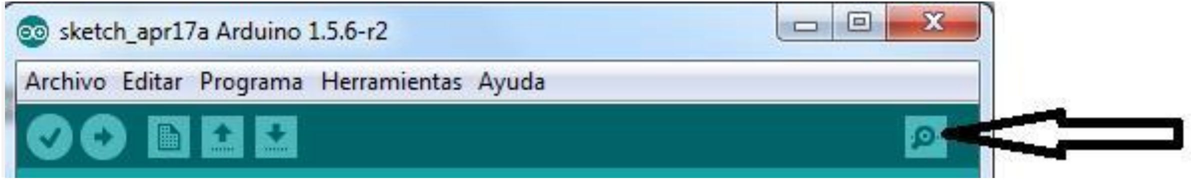
Nótese que Serial tiene la S mayúsculas y que C++ diferencia entre mayúsculas y minúsculas

La velocidad es una valor entre 300 y 115.200 bits por segundo. Y suele ser costumbre establecerla en 9600 (el valor por defecto) pero no hay ninguna razón para ello y esta no es una velocidad especialmente alta.

Para enviar un mensaje desde Arduino a nuestro PC podemos usar las funciones Serial.print() y Serial.println().Veamos un ejemplo:

```
int LED = 10 ; int boton = 6 ;
bool estado = false ;
void setup()
    {
       Serial.begin(9600) ; // Inicializa el Puerto seria 9600 bits por segundo
     }
void loop()
    {
        int i = 54 ;
        Serial.println( i );
    }
```

El println() enviara el valor de i al puerto serie de Arduino (repetidamente). Para leerlo en nuestro PC necesitamos un monitor de puerto serie. El IDE de Arduino incluye uno muy sencillo, pero suficiente que se invoca con el botón del monitor:



Necesitamos además asegurarnos de que la velocidad de conexión es la misma en ambos extremos. Fíjate en la parte inferior derecha del monitor serie:

José E. Martín UPG 2017 ©

Normalmente la velocidad por defecto son los 9600 bits por segundo o baudios en los que hemos programado nuestra puerta serie, y si lo desplegáis, veréis las diferentes velocidades aceptables para Arduino.

Estrictamente hablando, bits por segundo y baudios no son exactamente lo mismo salvo bajo ciertas condiciones particulares que en Arduino se cumplen, por lo que aquí podemos usarlos como sinónimos.

En el mundo Arduino parece haber un acuerdo de usar velocidades bajas como 9600 en lugar de más altas como 115200, para evitar problemas. Esto es algo que hace años estaba justificado por problemas de transmisión, pero con la tecnología actual no hay motivo para ello. Es más, en cuanto necesitemos utilizar dispositivos de comunicaciones como adaptadores Ethernet o BlueTooth para comunicarnos, la velocidad tendrá que subir necesariamente.

Ahora que sabemos enviar información y resultados al PC, vamos a ver cómo podemos operar con enteros y mostrar el resultado en la puerta serie. En C++ los operadores numéricos son los normales en cálculo (y algunos menos frecuentes):

```
Adición:            +
Resta:              -
Multiplicación:     *
División entera:    /     Cociente sin decimales (puesto que operamos con enteros
Resto:              %     Devuelve el resto de una división.
```

En C++ tenemos que expresar las operaciones matemáticas en una sola línea y utilizar paréntesis para garantizar que se opera como necesitamos. Vamos con algunos ejemplos:

OPERACIÓN	RESULTADO	COMENTARIO
int i = 4 * 2	resultado = 8	
int i = 4 * 2 / 3	resultado = 2	Porque desprecia los decimales al ser entero
int i = 14 % 3	resultado = 2	El resto de 14 entre 3
Int i = 2 + 8 / 2	resultado = 6	Calcula primero la división.
Int i = (2+8) / 2	resultado = 5	El paréntesis fuerza a que se realice primero la suma

José E. Martín UPG 2017 ©

Dada una expresión, la precedencia de operadores indica que operaciones se realizaran antes y cuales después en función de su rango. Para los que se inician en C++ no es fácil saber que operadores tienen preferencia, por lo que es más seguro que ante la duda uséis paréntesis.

Los paréntesis fuerzan las operaciones de una forma clara y conviene utilizarlos ante la duda porque de otro modo, detectar los errores de operación puede volverse muy difícil especialmente cuando uno empieza a programar.

El operador resto es más útil de lo que parece a primera vista porque nos permite saber si un numero es múltiplo de otro. Supongamos que queremos saber si un número dado es par.

Podríamos escribir un programa como este:

```
void setup()
     {
        Serial.begin(9600) ; // Inicializa el Puerto serie
     }
void loop()
    {
       int i = 27 ; //El número en cuestión
       if ( i % 2 == 0)
            Serial.println("Es par.") ;
      else
            Serial.println("Es impar");
    }
```

Dando a i distintos valores podemos comprobar cómo funciona el operador resto %. Volveremos sobre esto cuando veamos algunos ejemplos de cómo calcular números primos.

En este programa hemos usado de un modo diferente el Serial.println() pasándole una String de texto entrecomillada. Serial.print() envía el texto (entrecomillado) que le pongamos pero no da salto de línea cuando termina. En cambio Serial.println() hace lo mismo e incluye al final ese salto de línea.

```
void setup()
     {
        Serial.begin(9600) ; // Inicializa el Puerto serie
     }
void loop()
     {
          Serial.print("Buenos ") ;
          Serial.print("Dias ") ;
          Serial.println("a todos.") ;
     }
```

José E. Martín UPG 2017 ©

C++ dispone de un tipo de variables llamadas Strings, capaces de contener textos. Podemos operar con ellas simplemente definiéndolas como cualquier otro tipo de C++:

```
void loop()
   {
      int resultado = 25 ;
      String s = " El resultado es: " ; // Nótese que la S de string es mayúscula.
      Serial.print( s) ;
      Serial.println( resultado);
   }
```

Un tipo String se define simplemente poniendo entre comillas dobles un texto, y se puede operar con ellas de una forma similar a como operamos con enteros. Prueba:

```
void loop()
    {
        String a = "hola " ;
        String b = "a todos." ;
        Serial.println( a + b);
    }
```

Y también podemos construir un String sobre la marcha así:

```
void loop()
    {
        int resultado = 25 ;
        String s = "El resultado es: " ;
        Serial.println( s + String( resultado ));
    }
```

Donde imprimimos el resultado de concatenar s String, y la conversión de un int a String (El operador + añade un String al final de otro).

8.4 RECIBIENDO MENSAJES A TRAVÉS DEL PUERTO SERIE

Hasta ahora solo hemos enviado mensajes desde Arduino hacia el PC, ¿Pero como recibimos mensajes en Arduino?

En primer lugar disponemos de una función llamada Serial.parseInt() que nos entrega lo que se escribe en el monitor serie convertido a entero:

```
void loop()
    {
        if (Serial.available() > 0)
            {
                int x = Serial.parseInt();
                Serial.println ( x) ;
            }
    }
```

José E. Martín UPG 2017 ©

Este programa simplemente recibe en x los números que nos tecleen en la consola (cuando pulsemos intro) y si es un texto, lo interpreta como cero.

Hemos utilizado otra función de Serial : Available() que es un booleano. Conviene por costumbre comprobar que antes de leer el puerto serie hay algo que nos han enviado. Si lo hay Available() es True y en caso contrario es False.

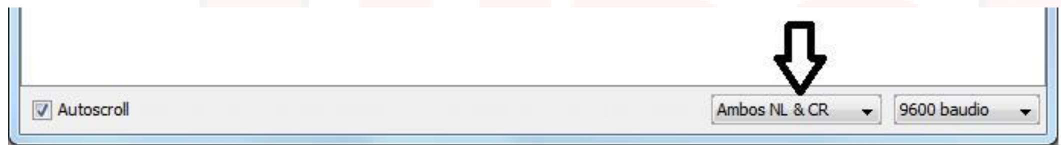
Para leer un String del puerto serie tenemos que complicarnos un poco más y hablar del tipo char.

Uno de de las mayores quebradero de cabeza al iniciarse en C++ es comprender la diferencia, anti-intuitiva, entre char y String. char es un tipo que representa un único carácter y se define con comillas simples, a diferencia de String que necesita comillas dobles:

```
char  c = ‘a’  ;
String s =”a”  ;
```

Aunque parezca lo mismo para C++ son muy distintos.

Para leer una cadena desde el puerto serie necesitamos leer un carácter cada vez y después montar un String a partir de ellos, pero antes, asegúrate de seleccionar ambos NL & CR en la parte inferior del monitor serie, para garantizar que se envía el carácter de fin de línea:



Un programa para leer la consola sería algo así:

```
    void setup()
           { Serial.begin(9600); }
    void loop ()
         {
             char c = ' ' ;
             String mensaje ="" ;
             if (Serial.available()) //Comprobamos si hay algo esperando
                  {
                     while( c != '\n') //Si lo hay, lo leemos hasta el intro
                         {
                           mensaje = mensaje + c ; // Añadimos lo leído al mensaje
                           c = Serial.read(); //Leer 1 carácter
                           delay(25);
                         }
                     Serial.println( mensaje); //Al salir imprimir el mensaje
                     mensaje = "" ; //Bórralo para la próxima vez
```


José E. Martín UPG 2017 ©

```
                    }
            }
```

Aquí usamos otra instrucción de C++ llamada while. Es similar a if, Ejecuta repetidamente el bloque que le sigue mientras se cumpla la condición que le pasamos entre paréntesis:

```
     while ( condición)
                  {   ……… }
```

Cuando lee el intro final de lo que escribimos, La condición c != '\n' se torna falso y sale del while.

Por lo demás, comprobamos si hay algo disponible en la puerta serie y de ser así montamos el mensaje leyendo un char cada vez y sumándoselo a mensaje para construir un String que podamos imprimir al salir.

El motivo del delay(25) es que a una velocidad tan lenta, enviar un char de 8 bits por la puerta serie, tarda mucho más de lo que tarda Arduino en ejecutar las instrucciones del while y volver a empezar. Por eso si se suprime el delay (y os recomiendo la prueba) leerá un carácter bueno (de la palabra escrita y como 10 caracteres basura para un Arduino UNO o Mega).

Si subimos la velocidad de comunicación a 115200 bits por segundo, comprobareis que no hay este problema ya que al multiplicar la velocidad de envío por más de 10 Arduino ya no tiene tiempo de volver a por más caracteres antes de que lleguen.

José E. Martín UPG 2017 ©

9. CREAR GRÁFICA UTILIZANDO EL PUERTO SERIE

9.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Un PC con el entorno de Arduino correctamente instalado y configurado.

9.2 DIBUJAR UNA GRÁFICA UTILIZANDO EL SERIAL PLOTTER

Ya hemos utilizado muchas veces el monitor serie del IDE para mostrar valores de las variables de nuestro programa. Pero en ciertas ocasiones puede ser útil verlo en forma de gráfica en vez de en datos numéricos, por ejemplo, para ver la tendencia de los datos de una forma sencilla y clara.

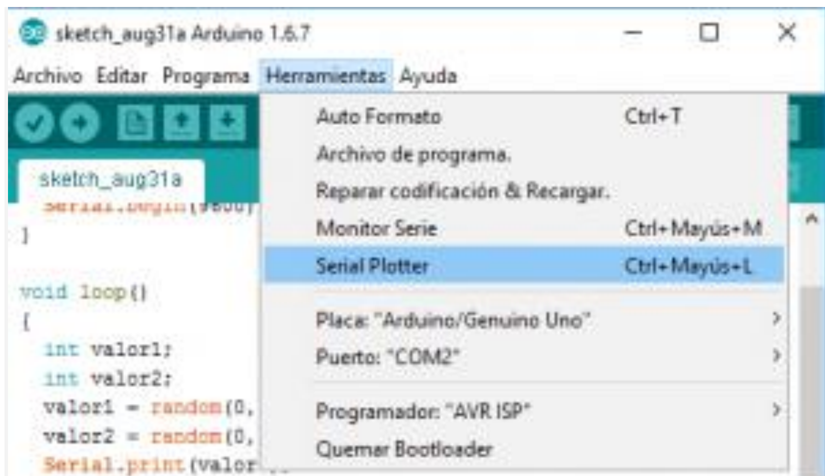
Pues resulta que el IDE de Arduino incorpora desde la versión 1.6.6 una herramienta llamada Serial Plotter que nos permite precisamente crear gráficas a partir de las variables que le indiquemos. Es muy sencillita de usar y por el momento no ofrece muchas opciones, pero seguramente vayan añadiendo características nuevas con nuevas versiones.

Para utilizarla no tenemos que aprender nada nuevo en cuanto a la programación. Simplemente haremos un programa que muestre un valor por el puerto serie. Podéis utilizar este programilla que genera números aleatorios cada cierto tiempo:

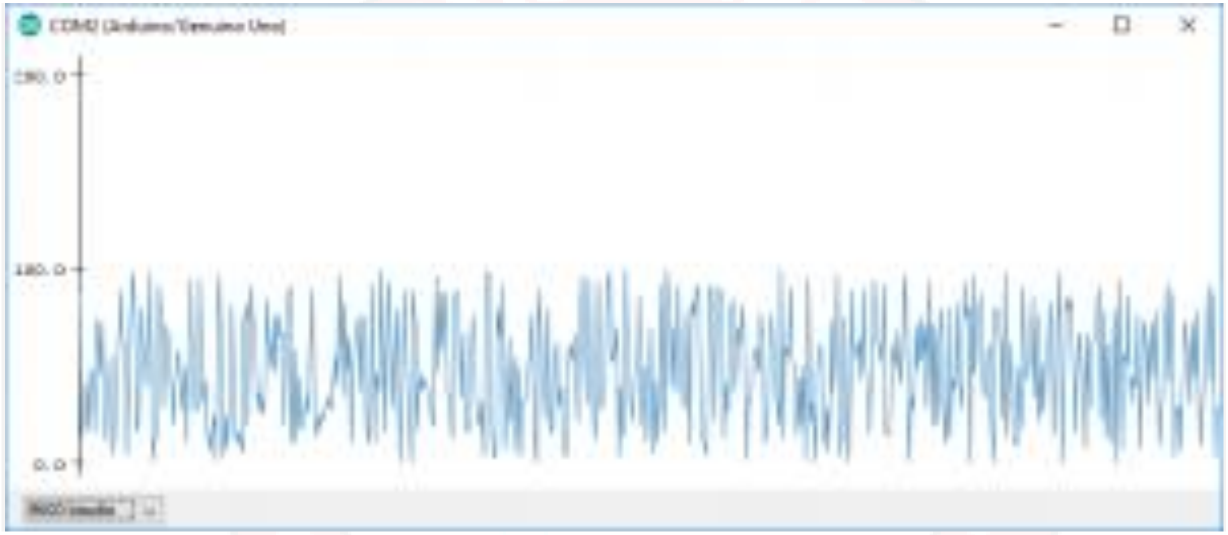
```
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  int valor;
  valor = random(0,100);
  Serial.println(valor);
  delay(250);
}
```

Ahora en vez de abrir el Monitor Serie, abriremos el Serial Plotter, que está en la barra de herramientas, justo debajo.

cc BY NC
José E. Martín UPG 2017 ©



Y cuando carguemos el programa en la placa, veremos cómo se va generando una gráfica a partir de los valores aleatorios que va cogiendo la variable.



9.3 CÓMO INCLUIR MÁS VARIABLES

La herramienta también nos da la opción de mostrar varias variables a la vez en la misma gráfica. La manera de hacerlo es también sencillísima, simplemente tendremos que sacar las mostrar las dos variables por el puerto serie pero utilizando un separador "," entre ellos, y automáticamente los dibujará en la misma gráfica con un color diferente.

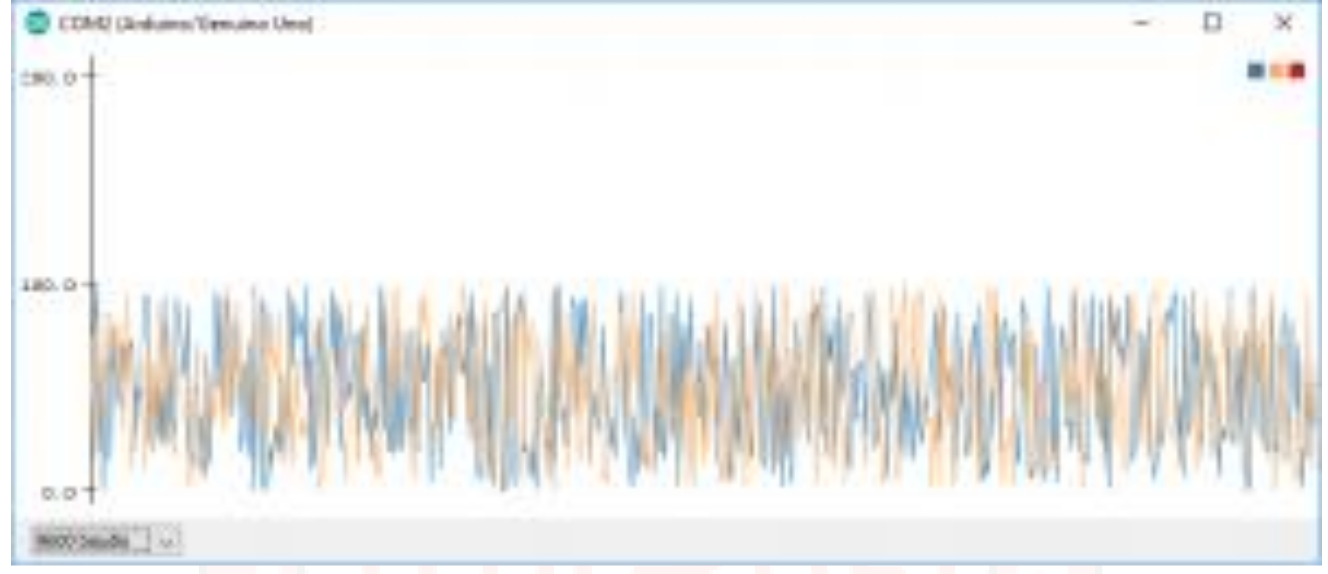
Si añadimos otra variable que tome también valores aleatorios al programa anterior, el programa quedaría de la siguiente forma: Grafica_2_variables.

```
void setup()
{
 Serial.begin(9600);
}

void loop()
{
 int valor1;
```

José E. Martín UPG 2017 ©

```
 int valor2;
 valor1 = random(0,100);
 valor2 = random(0,100);
 Serial.print(valor1);
 Serial.print(",");
 Serial.println(valor2);
 delay(250);
}
```



José E. Martín UPG 2017 ©

10. FUNCIONES Y ENTEROS.

10.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Un PC con el entorno de Arduino correctamente instalado y configurado.

10.2 LA PRIMERA FUNCIÓN: CALCULANDO SI UN NÚMERO ES PRIMO

Ya hemos comentado antes, que programar es un poco como andar en bici, se aprende pedaleando y a programar… programando. Hay que ir aprendiendo la sintaxis del lenguaje, C++ en nuestro caso, pero también aprendiendo a resolver problemas lógicos y partirlos en instrucciones.

Hacer cursos de programación (o de andar en bici) está bien, pero al final hay que ponerse a programar y tener problemas, porque solo teniéndolos y resolviéndolos, solo o con ayuda, se aprende. No se puede aprender a nadar sólo estudiando.

Con un cierto temblor de manos, vamos a centrarnos en esta sesión en algunos ejemplos clásicos de programación, como son el cálculo de números primos para entrenar esta capacidad de búsqueda de algoritmos prácticos para resolver problemas más o menos abstractos y para presentar algunos conceptos adicionales.

Es importante destacar que no existe una forma única de resolver un problema concreto y que una no tiene porque ser mejor que otra, aunque con frecuencia se aplican criterios de eficiencia o elegancia para seleccionar una solución.

Esta sesión va a requerir un esfuerzo un poco mayor que las anteriores porque vamos a empezar a entrenar un musculo poco usado, el cerebro, en una tarea poco frecuente, pensar. Y esto es algo que exige un poco e esfuerzo, pero es necesario para avanzar.

Supongamos que queremos crear un programa que nos devuelva true o false según que el número que le pasamos sea primo o no y a la que podamos llamar varias veces sin copiar el código una y otra vez. La llamaremos Primo () y queremos utilizarla de la siguiente manera: Si el numero n que le pasamos es primo nos tiene que devolver true y en caso contrario que devuelva false, o sea queremos que nos devuelva un valor bool.

Esto es lo que llamamos una función.

En realidad, ya hemos utilizado varias funciones que Arduino trae predefinidas como el Serial.print() o abs() , o Serial.available() y se las reconoce por esa apertura y cierre de paréntesis.

José E. Martín UPG 2017 ©

C++ nos ofrece todas las herramientas para crear nuestras propias funciones y es algo muy útil porque nos ayuda a organizar un problema general en trozos o funciones más pequeñas y más fáciles de manejar.

Para definir una función así, tenemos que declararla primero y describirle a C++ que hacer:

```
bool Primo( int x) // int x representa el parámetro que pasaremos a esta función
    {
        Aquí va lo que tiene que hacer
        …………
        return( bool);
    }
```

Declaramos la función Primo () como bool, o sea va a devolver un valor bool y por eso en algún punto tendremos que usar la instrucción return(true) o return(false) para devolver un resultado a quien la llame. Si devolviera un entero habría que definirla como int Primo(int x).

Si una función no va a devolver ningún valor, sino que simplemente realiza su trabajo y finaliza sin más entonces hay que declararla como void (vacía). Ya cononocemos dos funciones así : setup() y loop()

Veamos cómo podría ser el código de la función Primo():

```
  bool Primo( int n)
     {
        for ( int i = 2 ; i <n ; i++)
           {
             if ( n % i == 0) // Si el resto es 0 entonces es divisible.
                {
                  Serial.println ( String(n) +" es divisible por: " + String(i)) ;
                  return(false) ;
                }
           }
        return (true) ;
 }
```

Para saber si un número es o no primo basta con dividirlo por todos los números positivos menores que él y mayores que 1. En el ejemplo dividimos el número n empezando en 2 y finalizando en n-1.

Si encontramos un valor de i que devuelve resto 0, entonces es divisible (no es primo), devolvemos false con return y volvemos a la intruccion que llamo a la función. Si no hallamos ningún divisor, al finalizar el for devolvemos true y listo. Este es el método de fuerza bruta y sin duda es mejorable pero de momento nos sirve.

Para usar Primo hay que pasarle un entero. Recordad que al definir la función dijimos bool Primo (int n) donde n representa el valor que queremos probar. Así pues

José E. Martín UPG 2017 ©

```
    void loop() // Prog_8_1
        {
            int x = 427 ; // El número a probar
            bool p = Primo(x);
           if (p )
               Serial.print( String(x) + " Es primo.") ;
           else
               Serial.print( String(x) + " No es primo." ) ;
        }
```

Veamos cuantos primos hay hasta el, digamos 1024,

```
     bool control = true ; // Prog_8_2
     int maximo = 1024 ;
     void loop()
         {
             if ( control) // Solo es para que no repita una y otra vez lo mismo
                 {
                Serial.println( "Los numeros primos hasta el " + String( maximo)) ;
                      for ( int x = 2 ; x < maximo ; x++)
                            {
         bool p = Primo(x);
         if (p ) Serial.println( x) ; // No hay inconveniente en escribirlo seguido
                            }
                 }
             control = false ;
         }
     bool Primo( int n)
         {
             for ( int i = 2 ; i <n ; i++)
                  {
                       if ( n % i == 0) // Si el resto es 0 entonces es divisible.
                       return(false) ;
                  }
             return (true) ; // Si llega aqui es que no ha encontrado ningun divisor
         }
```

Aunque el programa funciona correctamente la salida no es muy presentable(Recordad que nos gusta ser elegantes). Vamos a formatearla. Para ello usaremos el carácter tabulador que se representa como '\t' y una coma después.

```
    bool control = true ; //Prog_8.3
    int maximo = 1024 ;
    int contador = 1 ;
```

José E. Martín UPG 2017 ©

```
void loop()
  {
    if ( control)    // Solo es para que no repita una y otra vez lo mismo
       {
             Serial.println( "Los numeros primos hasta el " + String( maximo)) ;
             for ( int x = 2 ; x < maximo ; x++)
                      {
                         if (Primo(x) )
                           if ( contador++ % 8 == 0)
                                Serial.println( String(x)+"," ) ;
                           else
                                Serial.print( String(x) +","+ '\t') ;
                      }
        }
          control = false ;
 }
```

Ahora el programa formatea la salida de una forma un poco más presentable y cómoda de leer.

```
COM7                                                    Enviar
Los numeros primos hasta el 1024
2,      3,      5,      7,      11,     13,     17,     19,
23,     29,     31,     37,     41,     43,     47,     53,
59,     61,     67,     71,     73,     79,     83,     89,
97,     101,    103,    107,    109,    113,    127,    131,
137,    139,    149,    151,    157,    163,    167,    173,
179,    181,    191,    193,    197,    199,    211,    223,
227,    229,    233,    239,    241,    251,    257,    263,
269,    271,    277,    281,    283,    293,    307,    311,
313,    317,    331,    337,    347,    349,    353,    359,
367,    373,    379,    383,    389,    397,    401,    409,
419,    421,    431,    433,    439,    443,    449,    457,
461,    463,    467,    479,    487,    491,    499,    503,
509,    521,    523,    541,    547,    557,    563,    569,
571,    577,    587,    593,    599,    601,    607,    613,
617,    619,    631,    641,    643,    647,    653,    659,
661,    673,    677,    683,    691,    701,    709,    719,
727,    733,    739,    743,    751,    757,    761,    769,
773,    787,    797,    809,    811,    821,    823,    827,
829,    839,    853,    857,    859,    863,    877,    881,
883,    887,    907,    911,    919,    929,    937,    941,
947,    953,    967,    971,    977,    983,    991,    997,
1009,   1013,   1019,   1021,
Autoscroll                              Ambos NL & CR    9600 baudio
```

Para conseguirlo, hemos añadido una coma y un tabulador a cada número excepto a uno de cada 8 que añadimos intro. También tenemos una línea que conviene comentar:

```
if ( contador++ % 8 == 0)
```

Cuando a una variable se le añaden dos símbolos más al nombre, significa que primero se use su valor actual en la instrucción en curso, ene este caso en el if, y *después* se incremente en 1 su valor.

Si hubiéramos escrito:

José E. Martín UPG 2017 ©

```
if ( ++contador % 8 == 0)
```

Quería decir que queremos incrementar su valor antes de utilizarlo. Esta notación es muy habitual en C++ y conviene reconocerla. También podemos usar contador- – y – -contador para decrementar.

10.3 EL TIPO ENTERO

Este sería un buen momento para preguntarnos hasta donde podría crecer máximo en el programa anterior. Le asignamos un valor de 1024, pero ¿Tiene un entero límite de tamaño?

La respuesta es afirmativa. Los enteros int en Arduino C++ utilizan 16 bits por lo que el máximo seria en principio 216 = 65.536, Pero como el tipo int usa signo, su valor está comprendido entre
-32.768 y +32.767.

De hecho en Arduino C++ hay varios tipos de distintos tamaños para manejar enteros:

TIPO	DESCRIPCIÓN	VALOR
int	Entero con signo, 16 bits	entre -32,768 y 32,767
unsigned int	Entero sin signo, 16 bits	2^{16} – 1 ; de 0 hasta 65.535
long	Entero con signo, 32 bits	2^{32} – 1 ,Desde -2.147.483,648 hasta 2.147.483.647
unsigned long	Entero sin signo, 32 bits	Desde 2^{32} – 1 ; 0 a 4.294.967.295
byte	Entero sin signo, 8 bits	2^{8} de 0 hasta 255

Todos estos tipos representan enteros con y sin signo y se pueden utilizar para trabajar con números realmente grandes pero no sin límite.

De hecho C++ tiene la fea costumbre de esperar que nosotros llevemos el cuidado de no pasarnos metiendo un valor que no cabe en una variable. Cuando esto ocurre se le llama desbordamiento (overflow) y C++ ignora olímpicamente el asunto, dando lugar a problemas difíciles de detectar si uno no anda con tiento.

Prueba este a calcular esto en un programa:

```
int i = 32767 ;
Serial.println ( i+1);
```

Enseguida veras que si i=32767 y le incrementamos en 1, para C++ el resultado es negativo. Eso es porque sencillamente no controla el desbordamiento. También es ilustrativo probar el resultado de

```
int i = 32767 ;
```

José E. Martín UPG 2017 ©

```
Serial.println (2*   i + 1);
Que según Arduino es -1.
```

Esto no es un error, sino que se decidió así en su día y C++ no controla los desbordamientos, así que mucho cuidado, porque este tipo de errores pueden ser muy complicados de detectar

10.4 MÁS SOBRE LAS FUNCIONES EN C++

Cuando se declara una función se debe especificar que parámetro va a devolver. Así:

Instrucción	Significa
int Funcion1()	Indica que va a devolver un entero
String Funcion2()	Indica que va a devolver un String.
unsigned long Funcion3()	Indica que va a devolver un long sin signo
void Funcion4()	No va a devolver valores en absoluto

Una función puede devolver cualquier tipo posible en C++, pero sólo puede devolver un único valor mediante la instrucción return(). Expresamente se impide devolver más de un parámetro. Si se requiere esto, existen otras soluciones que iremos viendo.

Este problema se puede resolver usando variables globales o pasando valores por referencia, y lo trataremos en futuras sesiones.

Lo que sí está permitido es pasar varios argumentos a una función:

```
int Funcion5 ( int x , String s , long y)
```

Aquí declaramos que vamos a pasar a Funcion5, tres argumentos en el orden definido, un entero un String y por ultimo un long.

cc BY NC
José E. Martín UPG 2017 ©

11. UN PROGRAMA CON VARIAS FUNCIONES.

11.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Un PC con el entorno de Arduino correctamente instalado y configurado.

11.2 PLANTEANDO UN PROGRAMA UN POCO MÁS COMPLICADO.

Hemos visto ya como definir funciones. En esta sesión vamos a plantear un programa que acepte un número desde la consola y compruebe si es o no primo. Y en caso de no serlo, nos calcule cuales son los divisores primos de este.

Normalmente para resolver un problema complejo es buena política partirlo en otros problemas más pequeños que podamos resolver más fácilmente. En este caso vamos a plantear al menos 3 funciones:

```
Primo() – Calcula si un número dado es primo, devuelve true y en caso contrario false.
Divisores() – Para un número dado nos devuelve sus divisores primos.
GetLine() – Vamos a definir una función genérica que recoja una cadena de texto de la puerta serie, para procesarla a posteriori. En este caso recogerá el número a probar.
```

La idea es, que nuestro programa empiece comprobando si un numero es primo. Si lo es, bien por el. SI no llamaremos a una función que calcule cuales son sus divisores. Y por ultimo necesitamos de algo que nos pueda dar un número desde la consola para probarlo y por eso escribimos otro programa que nos permita recibir cómodamente este numero de entrada.

Fijaros que casi sin haber escrito una linea de programa, ya he decidido como partirlo en bloques mas sencillos de manejar y programar. En otras palabras, he buscado una estrategia, de resolución

Esto es un poco trampa. Parece que se me acaba de ocurrir instantáneamente pero no (Aunque parezca increíble).

Después de pensar un rato y pasar un rato mayor escribiendo y afinando el programa, da gusto presentarlo como si fuera fácil. Que lo es, pero lo que no ves, es la de horas que hay que estar haciendo pruebas hasta todo va encajando y queda afinado.

Con tiempo y práctica (No, no mucha) iréis mejorando con rapidez, en vuestra habilidad de romper problemas en pedazos manejables, simplemente requiere tiempo y entrenamiento, y esto es algo que os será útil no solo para programar.

José E. Martín UPG 2017 ©

11.3 OPERANDO CON ARRAYS.

Con la función Primo() que vimos en la sesión anterior, a medida que el tamaño del número a probar, crece, el tiempo que tarda en determinar si es primo también, ya que dividimos por todos los números que le preceden.

Una manera más eficaz de calcular si un número es primo, es dividirlo solo por los números primos menores que el. Pero para esto necesitaríamos un modo de archivar estos primos.

Podríamos ejecutar primero el programa Prog_8.3 para hallar los N primeros números primos, y si dispusiéramos de algún medio para guardarlos, tendríamos un sistema más eficaz para decidir si un número es o no primo.

Una manera de archivar estos números es definir un array.

Un array es simplemente una colección de elementos organizados como una matriz, y pueden definirse con varias dimensiones. Empecemos con un array de una sola dimensión. Para definirlo podemos optar por dos maneras:

```
int serie1 [ 5] ; //Creamos una colección de 5 enteros
int serie2[] = { 3,5,6,12, 23} ;
```

En el primer caso definimos un array de enteros, de una sola dimensión con 5 elementos, sin asignar valores de momento.

En el segundo caso asignamos un array de enteros a los valores que le pasamos entre llaves, sin especificar cuantos, porque le dejamos a C++ la tarea de contarlos. Decimos que definimos el array por enumeración.

Para asignar o leer los valores de un array se utiliza un índice entre corchetes. Veamos este programa

```
int serie2[] = { 3,5,6,12, 23} ;            // Prog_9_1
void setup()
    {
        Serial.begin(9600) ;
    }
void loop()
    {
       for (int i=0 ; i<5 ; i++)
            Serial.println("Posicion " + String(i)+ ": "+ String(serie2[i])) ;
    }
```

El programa imprime el contenido del array recorriendo sus 5 posiciones.

Atención: la primera posición del un array es la 0 y la última el número de elementos – 1. Así serie2 [0] devuelve el primer elemento 3, y serie2[4] el último 23.

Un error muy peligroso, y difícil de detectar sería algo así (Prog_9.2):

```
int serie2[] = { 3,5,6,12, 23} ;
    for (int i=0 ; i<99 ; i++)
```

```
        Serial.println("Posicion " + String(i)+ ": "+ String(serie2[i])) ;
```

Uno esperaría que C++ generase un error, ya que definimos un array de 5 elementos y hacemos referencia a 100, pero no. Nuevamente C++ nos sorprende devolviendo correctamente los 5 primeros valores y luego sigue leyendo posiciones de memoria consecutivas tan tranquilo, como si tuvieran sentido.

C++ espera que seamos nosotros quienes controlemos esto, así que mucho cuidado

Por último, mencionar que podemos manejar arrays de varias dimensiones:

```
            Int Tablero[ 8, 8 ] ;
```

Imaginad que Tablero representa las posiciones de una partida de ajedrez y cada valor que contiene esa posición corresponde a una pieza que se encuentra en esa casilla.

11.4 AFINANDO LA FUNCIÓN PRIMO()

Si corremos el programa Prog_8.3 nos dará en el monitor una lista de primos hasta el 1024 (o hasta el número que deseemos modificando el valor de máximo) y seleccionándolos con el ratón podremos copiar esos valores y pegarlos en el IDE para crear un array con ellos (Prog_9.3):

```
int P[] =
{     2,       3,       5,       7,      11,      13,      17,      19,
     23,      29,      31,      37,      41,      43,      47,      53,
     59,      61,      67,      71,      73,      79,      83,      89,
     97,     101,     103,     107,     109,     113,     127,     131,
    137,     139,     149,     151,     157,     163,     167,     173,
    179,     181,     191,     193,     197,     199,     211,     223,
    227,     229,     233,     239,     241,     251,     257,     263,
    269,     271,     277,     281,     283,     293,     307,     311,
    313,     317,     331,     337,     347,     349,     353,     359,
    367,     373,     379,     383,     389,     397,     401,     409,
    419,     421,     431,     433,     439,     443,     449,     457,
    461,     463,     467,     479,     487,     491,     499,     503,
    509,     521,     523,     541,     547,     557,     563,     569,
    571,     577,     587,     593,     599,     601,     607,     613,
    617,     619,     631,     641,     643,     647,     653,     659,
    661,     673,     677,     683,     691,     701,     709,     719,
    727,     733,     739,     743,     751,     757,     761,     769,
    773,     787,     797,     809,     811,     821,     823,     827,
    829,     839,     853,     857,     859,     863,     877,     881,
    883,     887,     907,     911,     919,     929,     937,     941,
    947,     953,     967,     971,     977,     983,     991,     997,
   1009,    1013,    1019,    1021
 }  ;
```

Hemos definido un array enumerando sus elementos, entre llaves y separados por comas.

José E. Martín UPG 2017 ©

Es importante percatarse de que después de copiar y pegar las salida de Prog_8.3 hemos borrado la coma después del 1021, porque si no daría un error de sintaxis al definir el array.

Obsérvese que hay un punto y coma después de la llave de cierre del array. Aunque está entre llaves es una instrucción de asignación y no una definición de función.

Al definir el array por enumeración, si el número es alto podemos perder de vista cuantos elementos contiene. Si necesitamos calcular el número de miembros podemos utilizar la función sizeof():

```
int size = sizeof(P) / sizeof(int);
```

Donde P es nuestro array y dividimos por sizeof(int) porque definimos P como int. Y para este caso devuelve un valor de 172 elementos

Ahora bastaría dividir el número a probar por aquellos elementos del array P, menores que él:

```
bool Primo(int x)
{
    int index = 0 ;
    while ( P[index]  <  x)
        { if ( x  %  P[index++] == 0)
              return(false);
        }
    return(true);
}
```

Recorremos los valores almacenados en el array P[] mediante index al que vamos incrementando en cada iteración, hasta que nos toque probar un primo mayor que el valor que comprobamos.

11.5 FUNCIÓN DIVISORES ()

Esta función va a recorrer los elementos del array en tanto en cuanto sean menores (posibles divisores) que el número que probamos. Si encontramos divisores primos los guardamos en un array que hemos llamado Div[] al que le asignamos un máximo de 32 divisores:

```
int Div[32] ;
int Divisores(int x)
    {
        int index = 0 ;              //Apunta a la posicion del array P[]
        int pos = 0 ;                //Para apuntar al array de divisores Div[]
        while ( P[index] < x)
             {
                 int k = P[index++] ;
                 if ( x % k == 0)
```

José E. Martín UPG 2017 ©

```
                Div[pos++]= k ;       //Guardamos el divisor en en el array Div[].
             }                        // para uso posterior
        return(pos);                  //Devolvemos el numero de divisores encontrado
    }
```

Cuando queramos imprimir los divisores, basta con recorrer Div[].

Es importante entender que tanto la función Primo() como Divisores() recorren el array de números primos hasta que sobrepasan el valor del numero a probar. Si el número que probamos es mayor que el máximo primo que contiene P[], Podemos obtener resultados extraños, ya que leeremos más elementos de los que hemos definido.

Este método de buscar divisores es válido solo para números inferiores a 1024(o en su caso, al máximo número hasta el que hayamos calculado primos), porque un número no será primo si Primo() lo afirma, ya que encontrará divisores. Pero puede afirmar que un numero es primo erróneamente si sus divisores son superiores al máximo primo en P[].

11.6 LA FUNCIÓN GETLINE()

Aunque ya comentamos que podemos usar una función parseInt () incluida en Arduino para recoger un valor del puerto serie, tiene el inconveniente de que si no recibe una entrada salta al cabo de un tiempo (muy escasito) y devuelve 0, por lo que tendríamos que controlar el valor devuelto para que no se repitiese continuamente.

Por eso vamos a escribir una función de uso general que nos permita recoger una cadena de texto de la puerta serie sin que salga hasta que reciba un String que vamos a hacer finalice en intro.

```
String GetLine()
    {
        String S = "" ;
        if (Serial.available())
            {
                char c = Serial.read(); ;
                while ( c != '\n')           //Hasta que el character sea intro
                    {
                        S = S + c ;
                        delay(25) ;
                        c = Serial.read();
                    }
                return(S) ;
            }
    }
```

Definimos Getline() de tipo String, porque queremos que nos devuelva un texto. Comprobamos que hay algo disponible en la puerta serie, y en caso afirmativo construimos un String S añadiéndole cada uno de los caracteres que leemos del puerto serie, hasta que encontremos un intro.

José E. Martín UPG 2017 ©

Al encontrar el intro, se cumple la condición de salida del while y termina la función devolviendo la cadena construida (sin el intro).

Normalmente convendrá comprobar si hay algo disponible en la puerta serie antes de llamar a GetLine(), y si es así, la comprobación que hace GetLine() de tener algo disponible en el Serial seria redundante.

Pero si llamáramos a GetLine() sin comprobarlo y esta no lo controlase, quedaríamos atrapados en esta función hasta que alguien escribiera algo finalizado con intro para poder salir y podría no ser sencillo comprender el problema.

Nuevamente hemos incluido un delay de 25 ms en el while para asegurarnos de que Arduino no puede volver a leer mas caracteres antes de que a la velocidad de 9600 bps haya llegado el próximo carácter. Si la velocidad de comunicación es de 115200 bits por segundo o más, se puede suprimir este retraso.

11.7 EL PROGRAMA PRINCIPAL

Podemos ya escribir nuestra función principal loop(), que llame a las funciones que hemos definido a lo largo de esta sesión, para determinar si un numero que le pasamos por la puerta serie es primo o no y en caso negativo que nos muestre los divisores primos encontrados.

Podría ser algo así: Calculo de números primos en arduino:

```
void loop()
   {
     if (Serial.available())
      {
        String s = GetLine();
        int i = s.toInt() ; //Como esperamos un numero, convertimos el texto a numero
        if ( Primo(i))
           Serial.println(String(i) + " Es primo.");
        else
           {
              Serial.println(String(i) + " No es primo.");
              Serial.println("Sus divisores son: ");
              int j = Divisores(i);  //Recogemos el numero de divisores encontrados
              for (int n =0 ; n<j ; n++)     //Imprimimos los divisors del Div[]
              Serial.print(String(Div[n]) + ",\t");
              Serial.println(" ");   // Al acabar salta de linea
           }
      }
   }
```

Empezamos comprobando si hay algo sin leer en la puerta serie y si es así llamamos a GetLine() para que nos consiga lo que hay.

José E. Martín UPG 2017 ©

Como GetLine() nos devuelve un tipo String() usamos la función estándar de Arduino C++, s.toInt() que convierte el contenido String a tipo numérico int.

Después llamamos a Primo() para que compruebe este número. Si es primo, simplemente imprime un mensaje para confirmarlo. En caso contrario llamamos a Divisores() que busca y almacena en el array Div[] los divisores primos que encuentra.

Cuando divisores regresa, nos devuelve el número de divisores encontrados y podemos imprimirlos con un sencillo bucle for.

José E. Martín UPG 2017 ©

12. LOS PINES CUASI ANALÓGICOS.

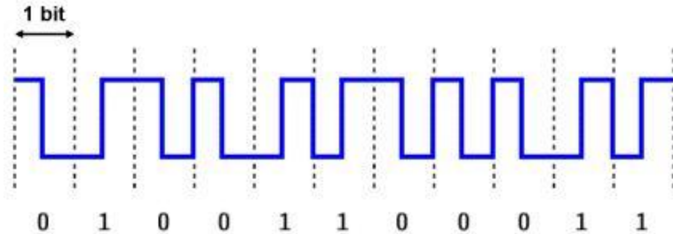
12.1 MATERIAL REQUERIDO.

	Arduino Uno o similar.
	Una Protoboard.
	Un diodo LED.
330Ω	na resistencia de 330 Ohmios..
	Algunos cables de Protoboard..

12.2 ANALÓGICO Y DIGITAL

Todas las señales que hemos manejado hasta ahora con nuestro Arduino, de entrada o de salida, comparten una característica común: Son digitales, es decir que pueden tomar un valor HIGH o LOW pero no valores intermedios.

Si representamos una el valor de una señal digital a lo largo del tiempo veríamos algo así:

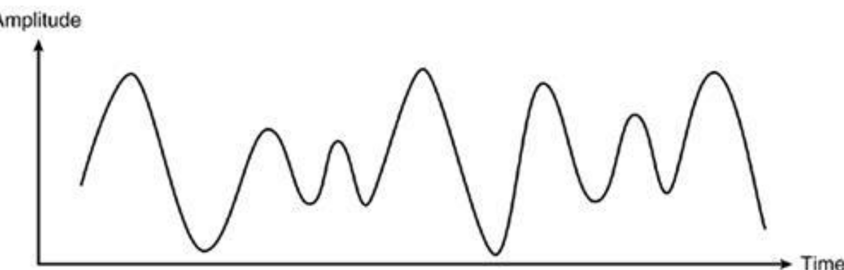


En la vida muchas cosas son así, apruebas o suspendes, enciendes la luz o la apagas, pero muchas otras son variables mensurables continuas y pueden tomar cualquier valor que

José E. Martín UPG 2017 ©

imaginemos, como el ángulo del reloj o la temperatura, que aun dentro de valores finitos pueden tomar tantos valores intermedios como podamos imaginar,

A esta clase de variables las llamamos analógicas y una representación por contraposición a lo digital, sería algo como esto:



No es raro que queramos controlar algo del mundo exterior con una señal analógica de forma que el comportamiento del sistema siga esa señal. Podemos por ejemplo querer variar la luminosidad de un diodo LED y no simplemente apagarlo o encenderlo

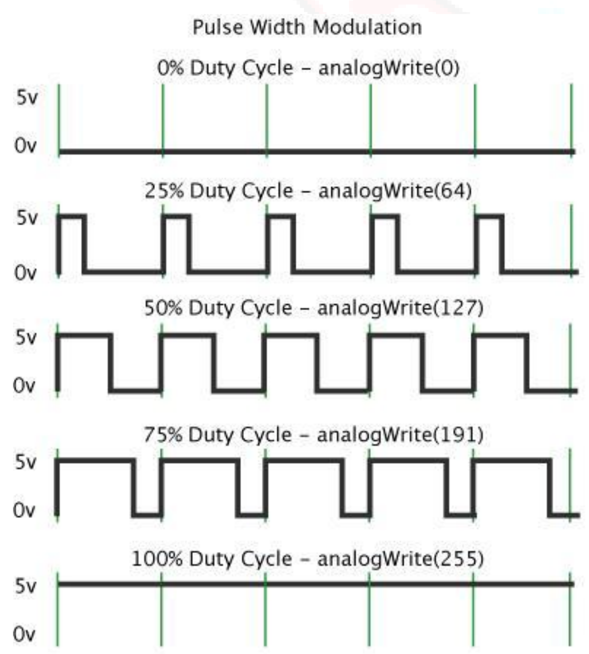
En esta sesión aprenderemos a enviar señales analógicas a los pines de salida de Arduino.

12.3 SALIDAS CUASI ANALÓGICAS

Hasta ahora hemos visto como activar las salidas digitales de Arduino, para encender y apagar un LED por ejemplo. Pero no hemos visto como modificar la intensidad del brillo de ese LED. Para ello, tenemos que modificar la tensión de salida de nuestro Arduino, o en otras palabras tenemos que poder presentar un valor analógico de salida.

Para empezar tenemos que dejar claro que los Arduino carecen de salidas analógicas puras que puedan hacer esto (con la notable excepción del Arduino DUE).

Pero como los chicos de Arduino son listos, decidieron emplear un truco, para que con una salida digital podamos conseguir que casi parezca una salida analógica.



A este truco se le llama PWM, siglas de Pulse Width Modulation, o modulación de ancho de pulsos. La idea básica es poner salidas digitales que varían de forma muy rápida de modo que el valor eficaz de la señal de salida sea equivalente a una señal analógica de menor voltaje.

Lo sorprendente es que el truco funciona.

Fijaros en la anchura del pulso cuadrado de arriba. Cuanto más ancho es, más tensión promedio hay presente entre los pines, y esto en el mundo exterior es equivalente a un valor analógico de tensión comprendido entre 0 y 5V. Al 50% es equivalente a una señal analogica del 50% de 5V, es decir 2,5. Si

José E. Martín UPG 2017 ©

mantenemos los 5V un 75% del tiempo, será el equivalente a una señal analógica de 75% de 5V = 3,75 V.

Para poder usar un pin digital de Arduino como salida analógica, lo declaramos en el Setup() igual que si fuera digital:

```
pinMode( 9, OUTPUT) ;
```

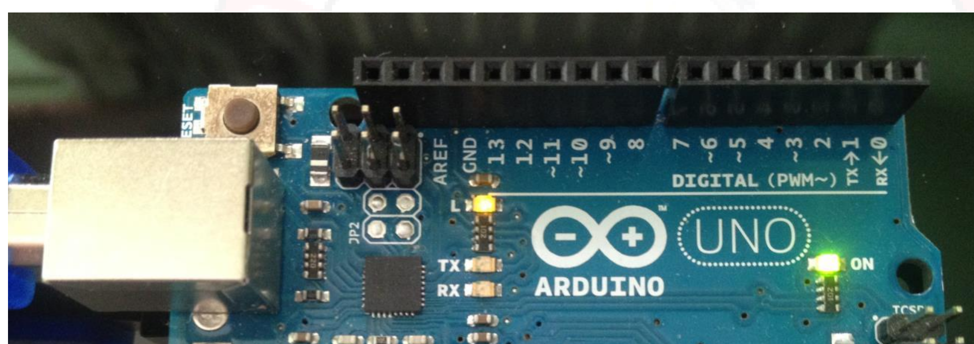
La diferencia viene a la hora de escribir en el pin:

```
digitalWrite(9, HIGH);                    //Pone 5V en la salida
digitalWrite(9, LOW);                     //Pone 0V en la salida
analogWrite( 9, V) ;
```

analogWrite escribe en el pin de salida un valor entre 0 y 5V, dependiendo de V (que debe estar entre 0 y 255).

De este modo si conectamos un LED a una de estas salidas PWM podemos modificar su brillo sin más que variar el valor que escribimos en el pin.

Pero hay una restricción. No todos los pines digitales de Arduino aceptan poner valores PWM en la salida. Solamente aquellos que tienen un símbolo ~ delante del número. Fijaros en la numeración de los pines de la imagen:



solo los que llevan simbolo

Solamente los pines 3, 5, 6, 9, 10 y 11 pueden hacer PWM y simular un valor analógico en su salida.

Si intentas hacer esto con un pin diferente, Arduino acepta la orden tranquilamente, sin error, pero para valores de 0 a 127 entiende que es LOW y para el resto pone HIGH y sigue con su vida satisfecho con el deber cumplido.

12.4 MODIFICANDO EL BRILLO DE UN LED

Vamos a hacer el típico montaje de una resistencia y un diodo LED, similar al de la sesión 3, pero asegurándonos de usar uno de los pines digitales que pueden dar señales PWM. En la imagen he usado el pin 9.

Podemos escribir un programa parecido a esto:

```
void setup()
   {
       pinMode( 9, OUTPUT) ;
```

José E. Martín UPG 2017 ©

```
    }
void loop()
    {
        for ( int i= 0 ; i<255 ; i++)
            {
                analogWrite (9, i) ;
                delay( 10);
            }
    }
```

El LED va aumentando el brillo hasta un máximo y vuelve a empezar bruscamente. Podemos modificar un poco el programa para que la transición sea menos violenta:

```
void setup()
    {
        pinMode( 9, OUTPUT) ;
    }
void loop()
{
        for ( int i= -255 ; i<255 ; i++)
              {
                  analogWrite (9, abs(i)) ;
                  delay( 10);
              }
}
```

Aquí aprovecho (por pura vagancia) para hacer el ciclo de subir y bajar el brillo del LED con un único bucle. La función abs(num), devuelve el valor absoluto o sin signo de un número num, y por eso mientras que i viaja de -255 a 255, abs(i) va de 255 a 0 y vuelta a subir a 255. ¿Que os parece el truco?

CC BY NC

José E. Martín UPG 2017 ©

13. LOS DIODOS LED RGB.

13.1 MATERIAL REQUERIDO.

	Arduino Uno o similar. Esta sesión acepta cualquier otro modelo de Arduino
	Una Protoboard.
	Un diodo LED RGB, independiente, o bien, con montura keyes.
330Ω	na resistencia de 330 Ohmios.
	Algunos cables de Protoboard.

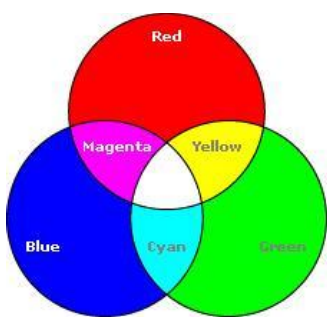
13.2 LOS DIODOS RGB

Hasta ahora hemos usado varias combinaciones de LEDS, pero siempre de un color definido. Habitualmente los rojos y amarillos son los más fáciles de conseguir, pero se pueden comprar también en tonos azules, verdes y hasta blancos. No suele haber grandes diferencias entre ellos excepto en el color.

Pero a veces es interesante disponer de una luz piloto que cambie de color según las condiciones. Por ejemplo, todos identificamos el verde como una señal de OK, mientras que el rojo indica problemas y el amarillo… bueno pues algo intermedio.

Poner varios diodos para hacer esto es engorroso y complica el diseño, así que estaría bien disponer de un diodo al que podamos indicar que color queremos que muestre. Esto es un LED RGB.

CC BY NC
José E. Martín UPG 2017 ©



Para quien este acostumbrado al diseño por ordenador ya está familiarizado con la idea de que podemos generar cualquier color en la pantalla con la mezcla, en diferentes grados de tres colores básicos:

Red : Rojo

Green: Verde

Blue: Azul

Es decir RGB, uno de esos acrónimos que surgen continuamente en imagen, TV, etc.

Un LED RGB es en realidad la unión de tres LEDs de los colores básicos, en un encapsulado común, compartiendo el Ground (cátodo es otro nombre más para el negativo).

En función de la tensión que pongamos en cada pin podemos conseguir la mezcla de color que deseemos con relativa sencillez

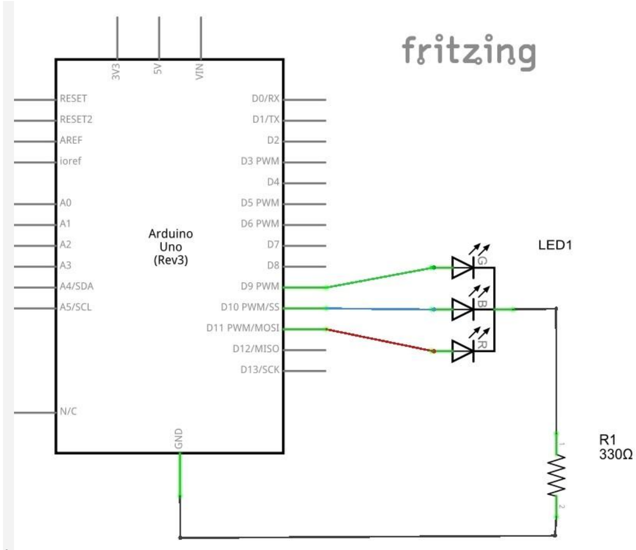
Para quien haya dibujado con lápices de colores o acuarelas, las mezclas de colores de arriba les resultará extraña. Esto es porque cuando pintamos en un papel blanco, la mezcla de colores es substractiva: Si mezclamos los tres colores obtenemos negro, o por lo menos algo oscuro

En cambio cuando pintamos con luz directamente, la mezcla es aditiva y obtenemos blanco al mezclar los tres colores básicos. Las reglas de mezcla de color en ambos casos son opuestas.

Vamos a montar un pequeño circuito que nos permita gobernar el color que emite uno de éstos LEDs de RGB.

cc BY NC
José E. Martín UPG 2017 ©

13.3 ESQUEMA DEL CIRCUITO



El montaje supone sencillamente conectar el negativo (el pin más largo) a Ground mediante una resistencia que limite la intensidad, y luego identificar los pines de colores:

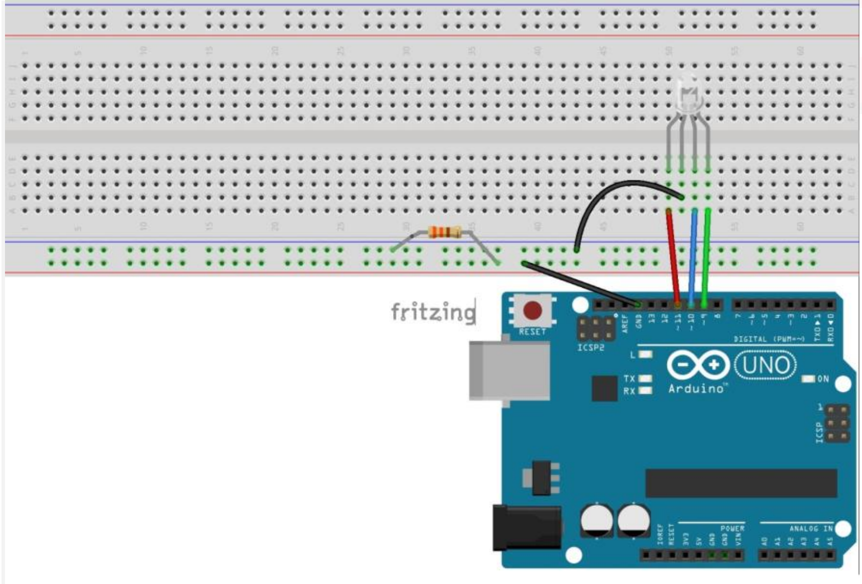
El pin más largo en estos LED es el GND.

Al lado de GND hay dos pines a un lado y uno solitario al otro. Por lo normal el solitario es el rojo R.

Así pues el pin out (patillaje) de un RGB LED suele ser R, GND, G, B.

De todos modos conviene asegurarse leyendo las especificaciones del fabricante, o bien identificando cada PIN. Para identificarlos basta conectar el GND a nuestro Arduino e ir probando cada una de las patas independientemente para ver qué color producen.

Si tu RGB tiene una montura Keyes, no tendrás que hacer esto, porque los pines vienen marcados y GND viene rotulado como -.



BY NC
José E. Martín UPG 2017 ©

Atención, en contra de la norma habitual, en este caso el cable rojo no indica la tensionVcc, sino el pin de gobierno del LED rojo.

En este esquema hemos utilizado los pines 9, 10 y 11. Podemos usar otros pero aseguraros de que puedan hacer PWM(los que tienen ~) para poder poner distintas intensidades.

13.4 PROGRAMA DE CONTROL RGB

Dado que nuestra idea es poder mezclar las tonalidades de los componentes RGB para generar diferentes matices de colores, parece buena idea escribir una función que haga esta mezcla de colores y a la que podamos recurrir de forma abstracta y práctica (además de para encapsular una utilidad curiosa, a la que podremos recurrir en futuros ejemplos y de paso insistir en el concepto de función).

Lo primero sería definir en el setup() los pines a usar:

```
void setup()
   {
       for (int i =9 ; i<12 ; i++)
            pinMode(i, OUTPUT);
   }
```

Y después podríamos escribir una función como esta

```
void Color(int R, int G, int B)
    {
        analogWrite(9 , R) ;   // Red    - Rojo
        analogWrite(10, G) ;   // Green - Verde
        analogWrite(11, B) ;   // Blue - Azul
    }
```

De este modo tendríamos fácil llamar a Color (0, 255, 0) para el verde. De hecho vamos a empezar asegurándonos de que tenemos identificados correctamente los pines, escribiendo un sketch como este:

```
void loop()
   {    Color(255 ,0 ,0) ;
        delay(500);
        Color(0,255 ,0) ;
        delay(500);
        Color(0 ,0 ,255) ;
        delay(500);
        Color(0,0,0);
        delay(1000);
   }
```

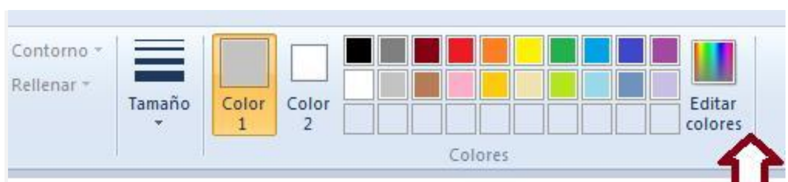
Este programa debería producir una secuencia de rojo, verde, azul, apagado y vuelta a empezar.

José E. Martín UPG 2017 ©

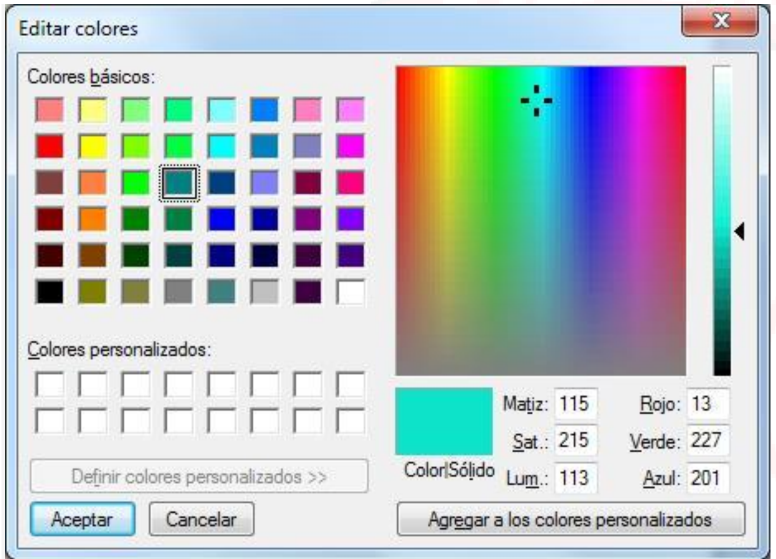
Conviene asegurarse de que hemos identificado correctamente los pines del RGB, porque de lo contrario, las mezclas posteriores de colores no serán lo que esperamos.

Vamos a ver como averiguar qué mezcla de RGB necesitamos para conseguir un color determinado. Para quienes uséis Windows disponéis del programa Paint incluido (en el menú de accesorios) y para quienes uséis Mac o Linux tenéis programas similares.

Si arrancáis el Paint(o equivalente) suele tener un selector de colores:



Pulsándolo aparecerá algo parecido a esto:



Si vais pinchando en la zona de colores de la derecha, en la barra vertical aparecen los matices próximos al que habéis pinchado y podéis elegir el que más os guste. Debajo podéis ver la separación en RGB precisa para conseguir un tono determinado.

Así pues para conseguir ese tono de azulito de la imagen basta con que llaméis a

```
Color(13, 227, 201) ;
```

Dado que Arduino nos permite escribir valores de 0 a 255 en los pines digitales, cuando utilizamos analogWrite(), en la práctica tendremos 255 x 255 x 255 colores diferentes o lo que es igual: 16.581.375 colores posibles.

La función Color() que hemos creado en esta sesión es muy sencilla pero se va añadiendo a otras que hemos ido creando en sesiones anteriores con lo que vamos haciendo una pequeña colección de ellas.

El grupo de desarrollo de Arduino ha ido creando también muchas funciones que están disponibles para incorporar en nuestros programas y que por razones de espacio resultan imposibles de ver más que muy por encima.

José E. Martín UPG 2017 ©

Solo como ejemplo introduciremos una de ellas. La función La función random(N) devuelve un valor al azar, comprendido entre 0 y N y en este caso, se presta especialmente bien para generar colores aleatorios en nuestro LED RGB. Probad esto:

```
    void setup()            //Prog_11_3
       {
          for (int i =9 ; i<12 ; i++)
               pinMode(i, OUTPUT);
       }
     void loop()
    {
          Color(random(255), random(255), random(255)) ;
          delay(500);
    }
void Color(int R, int G, int B)
    {
          analogWrite(9 , R) ;   // Rojo
          analogWrite(10, G) ;   // Green - Verde
          analogWrite(11, B) ;   // Blue - Azul
    }
```

Os generará un ciclo de colores aleatorios bastante psicodélico.

José E. Martín UPG 2017 ©

14. ARDUINO Y LAS PUERTAS ANALÓGICAS.

14.1 MATERIAL REQUERIDO.

	Arduino Uno o similar.
	Una Protoboard.
	Un diodo LED.
	Un potenciómetro de 10KΩ
330Ω	Una resistencia de 330 Ohmios.
	Algunos cables de Protoboard..

14.2 LOS POTENCIÓMETROS

Hasta ahora hemos usado siempre resistencias fijas, de un valor dado. Pero a veces es conveniente disponer de una señal variable para controlar el circuito que nos interesa. Imaginad el volumen de un equipo de música, o el dial que sintoniza una emisora en una radio FM.

Un potenciómetro es, simplemente, un mecanismo para proporcionar una resistencia variable.

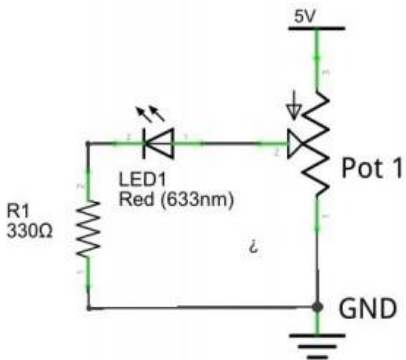
Hay potenciómetros de tantos tamaños, formas y colore,s como podáis imaginar, pero al final son una resistencia fija de un valor dado (10 kΩ en nuestro caso actual) y un

José E. Martín UPG 2017 ©

mecanismo que permita deslizar un dial conductor sobre esa resistencia, que nos permita tomar una parte de ese valor.

Por eso un potenciómetro siempre tiene 3 pines en fila. Los del extremo se comportan como una resistencia del valor de fondo de escala del potenciómetro, y un pin central que va tomando valores de resistencia en función del movimiento que hagamos con el ajuste.

Vamos a montar un circuito como este (en el que el potenciómetro esta rotulado Pot1):



La idea es conectar 5V y GND a los extremos del Potenciómetro (no importa cual es uno y otro) y luego conectar el pin central al positivo de un LED y el negativo a GND directo, pasando por una resistencia de limitación.

De este modo cuando giremos el potenciómetro estaremos modificando la tensión que aplicamos a la entrada del LED, que variara entre 0 y 5V (Aunque ahora parezca extraño es muy sencillo) y habremos conseguido un regulador de intensidad del LED.

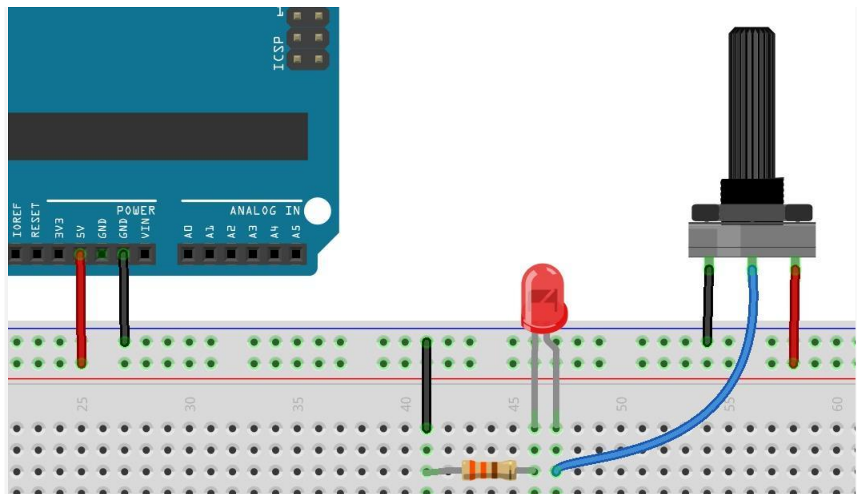
Con una resistencia de 10k la intensidad en el circuito será de: 5V / 10.000Ω = 0,5 mA Muy poco para conseguir iluminar el LED que requiere unos 20 mA. Así que durante la mayor parte del giro del potenciómetro el LED estará apagado.

Importante: No olvides la resistencia R1.Aunque el potenciómetro limite la intensidad, hay un momento en que llegara a cero y ahí y tu LED fallecerá en acto de servicio.

14.3 CIRCUITO PARA PROTOBOARD

El montaje en la protoboard sería similar a esto ya que vamos a utilizar el Arduino simplemente para dar tensión al circuito y nada más, Veréis que la intensidad de la luz varia de forma continua al girar el potenciómetro.

CC BY NC José E. Martín UPG 2017 ©



Recuerda que debido al exceso de resistencia del potenciómetro de prueba, durante la mayor parte del giro del ajuste el LED estará apagado.

Nótese que en este caso utilizamos nuestro Arduino simplemente como fuente de alimentación para dar tensión al circuito.

14.4 ARDUINO Y LAS ENTRADAS ANALÓGICAS

Con Arduino hemos visto que podemos influir en el mundo exterior aplicando salidas todo / nada en los pines digitales y también que usando PWM podemos simular bastante satisfactoriamente señales analógicas en algunos de esos pines.

También hemos visto cómo detectar pulsaciones de botones, definiendo como entradas los pines digitales. Pero en muchas ocasiones los sensores que usamos para supervisar el mundo exterior, nos entregan una señal analógica. Es el caso de los sensores de temperatura o distancia, de presión o PH, de intensidad de corriente en un circuito o de caudal de agua en una tubería.

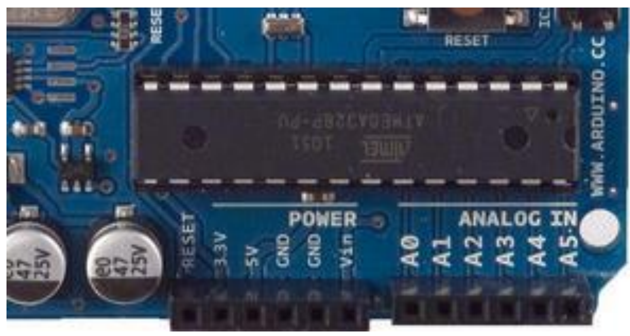
Para leer este tipo de señales continuas necesitamos un convertidor analógico a digital (o ADC por sus siglas en ingles) y que nos permite leer el valor de una señal analógica en un momento dado.

Estos convertidores toman una muestra del valor actual de la señal y nos entregan su valor instantáneo, medido en Voltios.

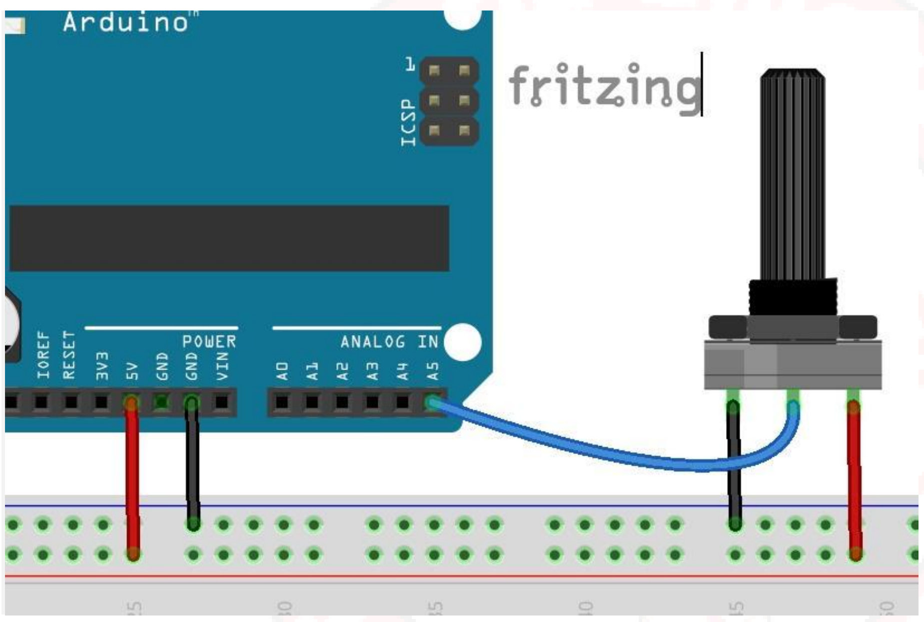
Mediante la lectura repetida de muestras a lo largo del tiempo podemos reconstruir la señal original con mayor o menor precisión, dependiendo de la exactitud de nuestra medida y de la velocidad a la que pueda tomar esas muestras.

José E. Martín UPG 2017 ©

Arduino UNO dispone de seis convertidores analógico a digital, nominados de A0 hasta A5, rotuladas como ANALOG IN:



Veamos cómo usar las entradas analógicas con un circuito como este, en el que damos tensión a los extremos de un potenciómetro y conectamos el pin central (el variable) a la entrada de la puerta A5 de Arduino:



Parece buen momento para destacar que los convertidores ADC leen valores de tensión y no resistencia, por lo tanto, lo que vamos a leer es la caída de tensión en el potenciómetro a medida que giramos el ajuste.

La primera curiosidad es que no necesitamos declarar en el setup() que vamos a usar una puerta analógica. Y la segunda es que para tomar una muestra (leer) del pin A5, usaremos la instrucción:

```
int Val = analogRead(A5) ;
```

Los convertidores de Arduino UNO y Mega son de 10 bits de resolución por lo que nos devolverá valores entre 0 y 2^{10} = 1.024 para tensiones entre 0 y 5V. En cambio el Arduino DUE dispone de convertidores de 12 bits por lo que el valor de sus lecturas estará entre 0 y 10^{12} o sea 4.096, es decir tiene mejor resolución (pero sólo puede leer hasta 3,3V).

Asegúrate de no usar sensores que puedan dar más de 5V máximo (con Arduino UNO y Mega), ya que dañarías el chip principal de Arduino.

José E. Martín UPG 2017 ©

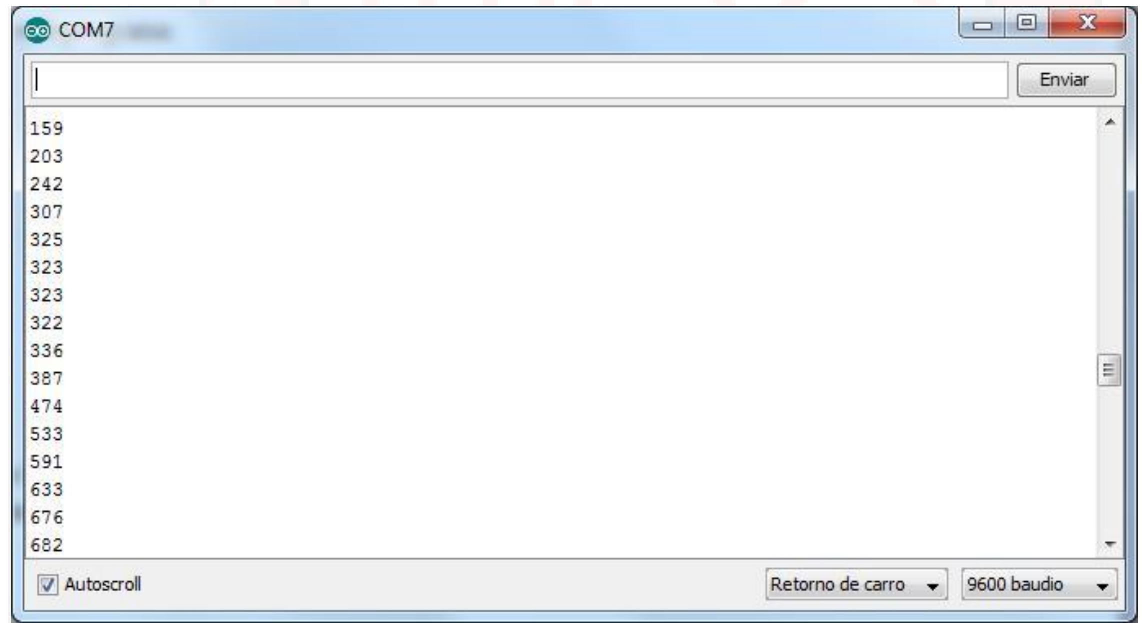
Vamos a escribir un programa que lea el valor del pin A5 y lo envíe a la consola para que podamos visualizarlo.

14.5 USANDO LAS PUERTAS ANALÓGICAS

Prueba este programa:

```
void setup()
    {
       Serial.begin(9600);    // Iniciamos la puerta serie
    }
void loop()
    {
       int Lectura = analogRead(A5) ;
       Serial.println( Lectura);
       delay(200) ;
    }
```

Cuando lo vuelques, arranca la consola y veras que a medida que giras el ajuste las lecturas varían de forma continua reflejando la posición del potenciómetro, las lecturas reflejan la caida en voltios en el.



No puedo resistirme a proponeros esta prueba: Desconecta el potenciómetro de la puerta A5 y observa los resultados que arduino envía a la consola. ¿Porque salen esos valores?

Al no estar el A5 conectado a ninguna referencia válida, está flotando y los valores que captura son muestra de esa incoherencia. En realidad lo que está haciendo tu Duino es captar ruido aleatorio de radiofrecuencia e intentar darle sentido, pero lo tiene mal, como podeis ver.

cc BY NC
José E. Martín UPG 2017 ©

No obstante en condiciones normales los valores que leerá seran relativamente bajos.¿Quieres que las oscilaciones crezcan en valor?. Fácil. Ponle una antena. Vale un simple cable de protoboard conectado desde el A5 a nada (O si coges el otro extremo entre los dedos, tu mismo haras de antena). Acabas de construir el receptor de Radio frecuencia mas inutil del mundo

14.6 UN ÚLTIMO COMENTARIO

Decíamos en una sección anterior, que la fidelidad con que podemos muestrear una señal analógica dependía, básicamente, de la resolución de la muestra y de la velocidad a la que podíamos muestrear la señal (Sample Rate en inglés).

Ya dijimos que la familia Arduino, dispone de convertidores de 10 bits por lo que nuestra resolución es de 2^{10} = 1.024 y en el caso del DUE de 2^{12} = 4.096. Pero hasta ahora no hemos visto a qué velocidad podemos tomar muestras con nuestro Arduino. Vamos a comprobarlo, con este mismo circuito.

Tenemos una función llamada millis() que nos indica en milisegundos el tiempo transcurrido desde que iniciamos Arduino y la podemos usar para ver cuantas muestras podemos tomar por segundo.

```
void setup()
   {    Serial.begin(9600); }
void loop()
   {     unsigned long T ;
         int n = 0 ;
         T = millis();
         while (millis() <= T + 1000)    // Mientras no pase un Segundo = 1000 mS
             {      analogRead( A5) ;
                    n++ ;                 // Contamos cada vez que leemos
             }
         Serial.println(n);
}
```

Hemos usado un unsigned long para guardar millis porque es el tipo que Arduino usa internamente para su reloj. Sería un error manejar millis con un int porque su valor máximo es 32.767 y midiendo milisegundos el contador desbordaría en poca más de 32 segundos.

SI corréis este programa en un Arduino UNO os dará, poco más o menos, un resultado de 8.940 muestras o lecturas por segundo. No está mal.

Es adecuado para muestrear señales que no varíen demasiado rápido con el tiempo, como son casi todos los sensores habituales en la industria, pero que se quedará corto si queréis muestrear señales de audio.

Para jugar con audio es mejor usar un Arduino DUE. Tiene una velocidad de reloj 4 veces más rápida(os hará falta), capacidad de muestreo a velocidad de audio (40Khz) y auténticos convertidores DAC (digital to analog converters).

José E. Martín UPG 2017 ©

De hecho no es complicado aumentar la velocidad de muestreo hasta unas 20.000 muestras por segundo con un Arduino UNO, pero para eso tenemos que puentear Arduino y saltar a programar el chip interior Atmega 328. No es momento para ello, pero hay formas.

BY NC
José E. Martín UPG 2017 ©