PRACTICAS ARDUINO - DOCUMENTACIÓN

ÍNDICE

1 INTRODUCCIÓN

1.1 ¿Qué es Arduino? [1]

Arduino es una plataforma open source, basada en una placa con un microcontrolador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

Con un Arduino, se puede tomar información del entorno a través de sus pines de entrada de toda una gama de sensores y puede actuar o controlar toda serie de dispositivos como: luces, motores, relés, etc, a través de sus pines de salida.

1.2 ¿Qué es un microcontrolador? [2]

Un **microcontrolador** (abreviado **µC**, **UC** o **MCU**) es un circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria. Está compuesto de varios bloques funcionales, los cuales cumplen una tarea específica. Un microcontrolador incluye en su interior las tres principales unidades funcionales de una computadora: unidad central de procesamiento, memoria y periféricos de entrada/salida.

1.3 Open source.

Al ser open-hardware, tanto su diseño como su distribución son libre. Es decir, puede utilizarse libremente para el desarrollo de cualquier tipo de proyecto sin haber adquirido ninguna licencia. Los planos para los módulos están publicados bajo licencia Creative Commons **[3],** por lo que diseñadores experimentados de circuitos pueden hacer su propia versión del módulo, extendiéndolo y mejorándolo. Incluso usuarios relativamente inexpertos pueden construir la versión de la placa del módulo para entender cómo funciona y ahorrar dinero.

Además no solo su hardware es abierto, también su software, es software libre porque se publica con una combinación de la licencia GPL (para el entorno visual de programación propiamente dicho) y la licencia LGPL (para los códigos fuente de gestión y control del microcontrolador a nivel más interno). La consecuencia de esto es, en pocas palabras, que cualquier persona que quiera (y sepa), puede formar parte del desarrollo del software Arduino y contribuir así a mejorar dicho software, aportando nuevas características, sugiriendo ideas de nuevas funcionalidades, compartiendo soluciones a posibles errores existentes, etc. **Esta manera de funcionar provoca la creación espontánea de una comunidad de personas que colaboran mutuamente a través de Internet**, y consigue que el software Arduino evolucione según lo que la propia comunidad decida. Esto va mucho más allá de la simple cuestión de si el software Arduino es gratis o no, porque el usuario deja de ser un sujeto pasivo para pasar a ser (si quiere) un sujeto activo y partícipe del proyecto.

1.4 Entorno de desarrollo (IDE)

Para que el microcontrolador cumpla sus funciones y “controle algo”, es necesario crear un programa y colocarlo en el microcontrolador. Para crear ese código fuente se necesita un compilador y otros elementos, que se integran en el IDE (integrated developmet environment).

Un IDE, es simplemente una forma de llamar al conjunto de herramientas software que permite a los programadores poder desarrollar (escribir y probar) sus propios programas con comodidad. En el caso de Arduino, su IDE nos permite escribir y editar nuestro programa (también llamado “sketch”), nos permite también comprobar que no hayamos cometido ningún error en el código y que además nos permite, cuando ya estemos seguros de que el sketch es correcto, grabarlo en la memoria del microcontrolador de la placa Arduino para que este se convierta a partir de entonces en el ejecutor autónomo de dicho programa.

2 HARDWARE

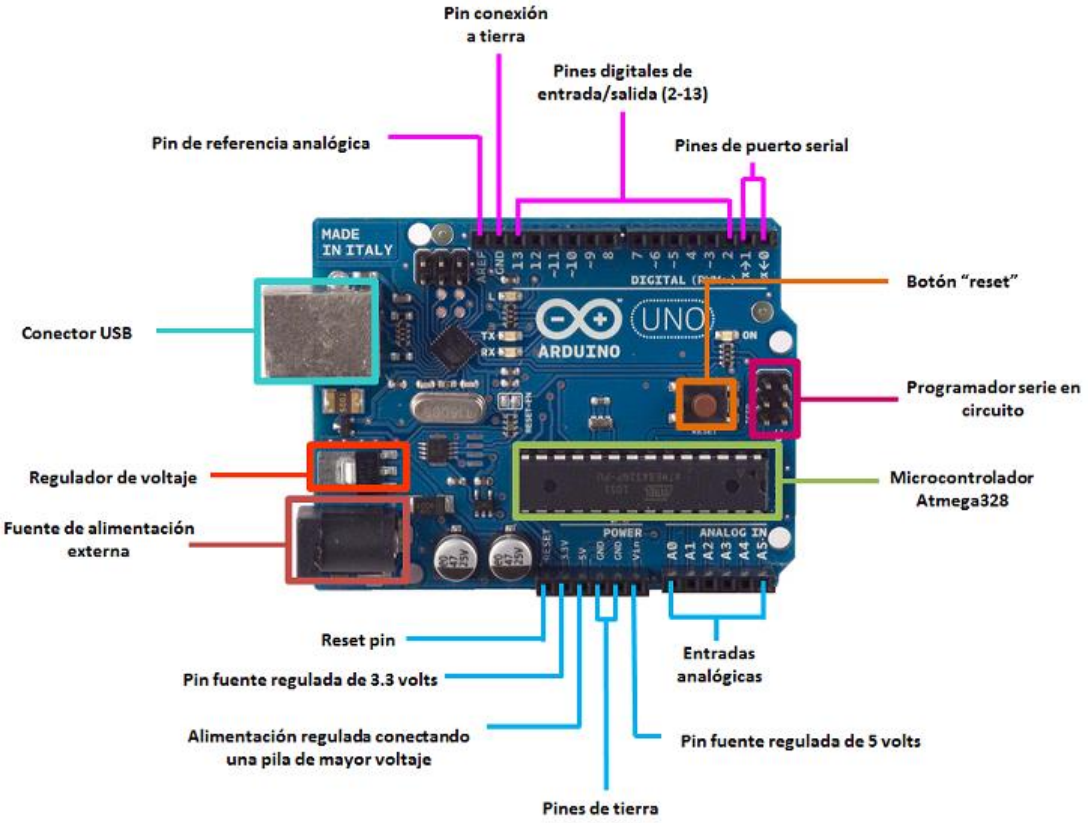


Ilustración 1

2.1 Microcontrolador

Dichos microcontroladores son de la compañía Atmel, con una arquitectura de tipo AVR, arquitectura desarrollada por Atmel y en cierta medida "competencia" de otras arquitecturas como por ejemplo la PIC del fabricante Microchip.

El microcontrolador, definirá las características de nuestra placa, ya que en función de este se definirá: la memoria flash del dispositivo (tamaño del programa) , EEPROM(que puede ser leída y escrita con la librería EEPROM1) y RAM; el número de entradas y salidas disponibles, la velocidad (velocidad de reloj máxima admitida), etc…

Hay varias placas de desarrollo, cada una con diferentes características **[4] [5],** erradicando su principal diferencia en el microcontrolador utilizado, (como hemos visto antes).

2.2 El gestor de arranque del microcontrolador.

Dentro de la memoria Flash del microcontrolador incluido en las placas Arduino viene pregrabado de fábrica un pequeño programa llamado "bootloader" o "gestor de arranque", que resulta imprescindible para un cómodo y fácil manejo de la placa en cuestión. Este software

(también llamado “firmware”, porque es un tipo de software que raramente se modifica) ocupa, en la placa Arduino UNO, 512 bytes de espacio en un apartado especial de la memoria Flash, el llamado “bootloader block”, pero en otros modelos de placas Arduino puede ocupar más (por ejemplo, en el modelo Leonardo ocupa 4 Kilobytes).

La función de este firmware es gestionar de forma automática el proceso de grabación en la memoria Flash del programa que queremos que el microcontrolador ejecute.

Más concretamente, el bootloader se encarga de recibir nuestro programa de parte del entorno de desarrollo Arduino (normalmente mediante una transmisión realizada a través de conexión USB desde el computador donde se está ejecutando dicho entorno hasta la placa) para proceder seguidamente a su correcto almacenamiento en la memoria Flash, todo ello de forma automática y sin que nos tengamos que preocupar de las interioridades electrónicas del proceso. Una vez realizado el proceso de grabación, el bootloader termina su ejecución y el microcontrolador se dispone a procesar de inmediato y de forma permanente (mientras esté encendido) las instrucciones recientemente grabadas.

Si adquirimos un microcontrolador ATmega328P por separado, hay que tener en cuenta que no dispondrá del bootloader, por lo que deberemos incorporarle uno nosotros “a mano” **[6]** para hacer uso de él a partir de entonces, o bien no utilizar nunca ningún bootloader y cargar entonces siempre nuestros programas a la memoria Flash directamente. En ambos casos, el procedimiento requiere el uso de un aparato específico (en concreto, lo que se llama un “programador ISP” –In System Programmer–), que debemos adquirir aparte o también podemos utilizar un Arduino para como programador, ya que el IDE de Arduino, nos permite cargar un programa sobre nuestra placa para utilizarlo como programador (Herramientas/Programador/Arduino as ISP) . Este aparato se ha de conectar por un lado a nuestro computador y por otro al chip (Atmega), y suple la ausencia de bootloader haciendo de intermediario entre nuestro entorno de desarrollo y la memoria Flash del microcontrolador. Por lo tanto, podemos resumir diciendo que el gestor de arranque es el elemento que permite programar nuestro Arduino directamente con un simple cable USB y nada más.

2.3 Interface de comunicación

Sirve como puente para la comunicación entre el microcontrolador y el puerto USB.

El Arduino tiene un número de infraestructuras para comunicarse con un ordenador, otro Arduino, u otros microcontroladores. El ATmega328 (arduino uno) provee comunicación serie UART TTL (5 V) **[7],** la cual está disponible en los pines digitales 0 (Rx) y 1 (Tx). Un FTDI FT232RL o un Atmega 16u (8u) (interface comentado anteriormente) en la placa canaliza esta comunicación serie al USB y los drivers FTDI (incluidos con el software Arduino) proporcionan un puerto de comunicación virtual al software del ordenador. El software Arduino incluye un monitor serie que permite enviar y recibir datos desde la placa Arduino.
Una libreria SoftwareSerial permite comunicación serie en cualquiera de los pines digitales, pero hemos de tener en cuenta que la velocidad de comunicación a través de estos será menor, ya que su electrónica no es específica para dicho fin, como sucede para los pines Rx y TX.

Para el caso de que la comunicación la proporcione el chip Atmega 16u(8u), al ser un micro-controlador, este contiene un programa (firmware), para proporcionar dicha comunicación.

Por eso, el Arduino viene provisto de un conector para la programación ISP, para poder actualizarle, modificarle o simplemente reinstalarle, ya que a veces puede estropearse y que sea necesario recargarlo.

Si hubiera que hacerlo: **[8] [9]**

2.4 Cristal

Un cristal oscilador a 16Mhz, El oscilador de cristal se caracteriza por su estabilidad de frecuencia y pureza de fase, dada por el resonador. La frecuencia es estable frente a variaciones de la tensión de alimentación. Su misión es establecer la velocidad de reloj del microcontrolador.

2.5 Alimentación

- **Vin.** Entrada de voltaje a la placa Arduino cuando se está usando una fuente externa de alimentación. Este pin esta interconectado con el conector (Jack) destinado para dicho fin. La tensiones de entrada para dicho pin, estará entre 7 y 12V, tensión recomendada, aunque admite tensiones entre 6 y 20V. Esta tensión es regulada y transformada por un regulador de voltaje **[10],** para dar los 5V de funcionamiento del microcontrolador.

- **5V.** Voltaje estabilizado usado para alimentar el microcontrolador y otros componentes de la placa. Esta puede provenir de VIN a través de un regulador integrado en la placa, (intensidad máxima de 300 mA), o proporcionada directamente por el USB u otra fuente estabilizada de 5V.

- **3.3V.** Proporciona una tensión de 3,3 V, y una intensidad máxima de 50 mA.

- **GND.** Es masa, toma de tierra, o nivel 0 V de referencia. Es muy importante que todos los componentes de nuestros circuitos compartan una tierra común como referencia.

2.6 Protección de Sobrecarga del USB.

El Arduino tiene un fusible reseteable que protege los puertos USB del ordenador de cortes y sobrecargas. Aunque la mayoría de los ordenadores proporcionan su propia protección interna, el fusible proporciona una capa de protección extra. Si más de 500 mA se aplican al puerto USB, el fusible automáticamente romperá la conexión hasta que el corte o la sobrecarga sean eliminados.

2.7 Botón de Reset y Reset Pin.

Sirve para suministrar un valor LOW (0V) para reiniciar el microcontrolador. El Pin Reset, está destinado para añadir un botón de Reset en los shields , que no dejan acceso al botón de la placa.

2.8 ICSP (ISP)

Conector para la programación ICSP (In Circuit Serial Programming, o Programación Serial en circuito).

Es un método para programar directamente microcontroladores de tipo AVR, PIC y Parallax Propeller que no tienen el bootloader preinstalado. (Ver apartado 2.2)

2.9 AREF

Tensión de referencia para las entradas analógicas. Se utiliza con analogReference () **[11].**

Sirve básicamente para establecer una tensión de referencia, para las lecturas analógicas. Los valores podrán ser distintos según el chip, el valor por defecto será 5V.

2.10 Entradas y salidas digitales

Los pines digitales, pueden ser usados como entradas o salidas,(usando funciones pinMode(), digitalWrite() y digitalRead()). Operan a 5 voltios. Cada pin puede proporcionar o recibir un máximo de 40 mA y tiene una resistencia interna _pull-up_ (desconectada por defecto) de 20-50 KOhms.

Muchas de estas salidas digitales, tienen otras funciones especiales, pudiendo funcionar también como salidas PWM, comunicación Rx, TX , como pines para la comunicación ISP, comunicación I2C ó como interruptores externos, en función de su electrónica interna y del tipo de chip utilizado.

2.11 PWM

Ciertas de las entradas y salidas digitales, pueden trabajar como salidas de PWM. Estas están marcadas con el símbolo ~ .

Los PWM, son salidas digitales que tienen la peculiaridad de que pueden comportarse o simular una salida analógica, gracias al envío de pulsos de distinta duración. El sistema PWM usa una frecuencia constante, que en los Arduinos suele ser de 490HZ . Estos pulsos se pueden regular su ancho, desde 0-255 (8 bits) representando el rango de voltaje de la placa. (0-5V).

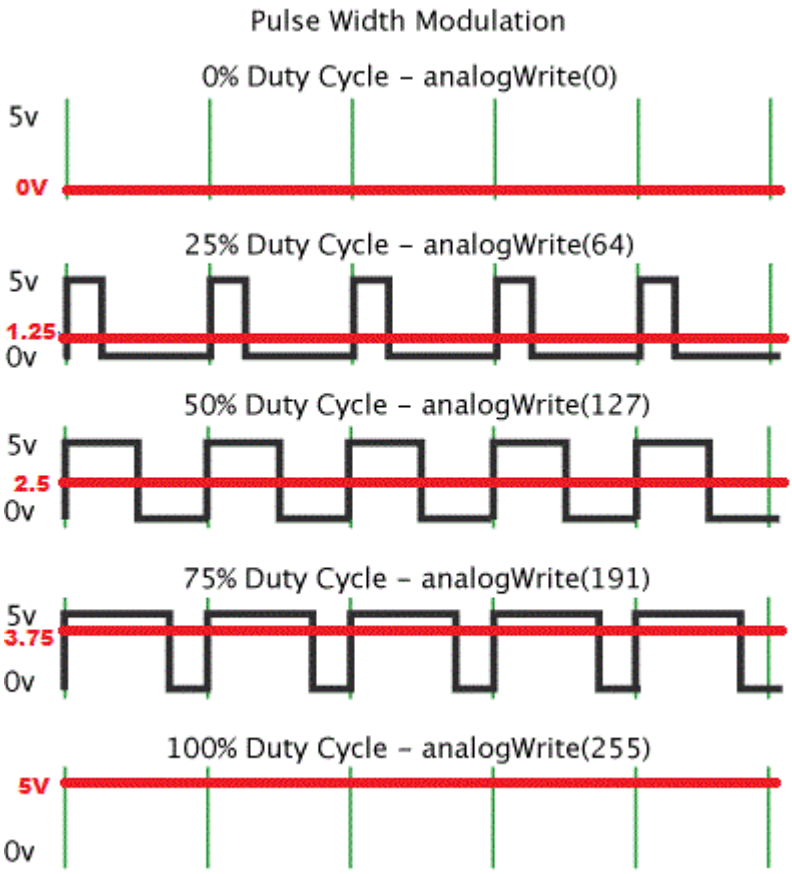


Ilustración 2

2.12 Serial: Rx y Tx.

Usados para recibir (Rx) y transmitir (Tx) datos TTL en serie. Estos pines, están conectados a los pines correspondientes del chip dedicado a la comunicación USB-a-TTL Serie.

2.13 Interruptores externos:

2 y 3. Estos pines pueden ser configurados para disparar un interruptor mediante un valor bajo, un margen creciente o decreciente, o un cambio de valor. Mirar la función attachInterrupt().

2.14 SPI

SS (10), MOSI (11), MISO (12), SCK (13) (pines correspondientes al Arduino Uno). Estos pines soportan comunicación SPI, la cual, aunque proporcionada por el hardware subyacente, no está actualmente incluida en el lenguaje Arduino.

2.15 I²C [12]

SDA (4) y SCL (5) (pines correspondientes al Arduino Uno). Soportan comunicación I²C (TWI) usando la librería Wire.

I²C es un bus de comunicaciones en serie. Su nombre viene de *Inter-Integrated Circuit* (Inter-Circuitos Integrados)

Es un bus muy usado en la industria, principalmente para comunicar microcontroladores y sus periféricos en sistemas integrados (*Embedded Systems*) y generalizando más para comunicar circuitos integrados entre sí que normalmente residen en un mismo circuito impreso.

La principal característica de **I²C** es que utiliza dos líneas para transmitir la información: una para los datos y otra para la señal de reloj. También es necesaria una tercera línea, pero esta sólo es la referencia (masa).

2.16 Entradas analógicas

El Uno tiene 6 entradas analógicas, y cada una de ellas proporciona una resolución de 10bits (1024 valores). Por defecto se mide de tierra a 5 voltios, aunque es posible cambiar la cota superior de este rango usando el pin AREF y la función analogReference().

3 Software

3.1 Instalación de Arduino. [13]

Instalación: instrucciones paso a paso para configurar el software de Arduino y conectarlo a una placa Arduino.

1. Vamos a la página de descargas de Arduino http://arduino.cc/en/Main/Software
2. Y descargamos la versión comprimida en ZIP para nuestro Windows.
3. Una vez que tenemos descargado el archivo lo descomprimimos, y obtendremos una carpeta con el nombre "arduino-*version del programa*" en nuestro caso "arduino-1.0.5-r2". Dentro de esta carpeta encontraremos un icono con el símbolo de arduino que nos abrirá la aplicación.

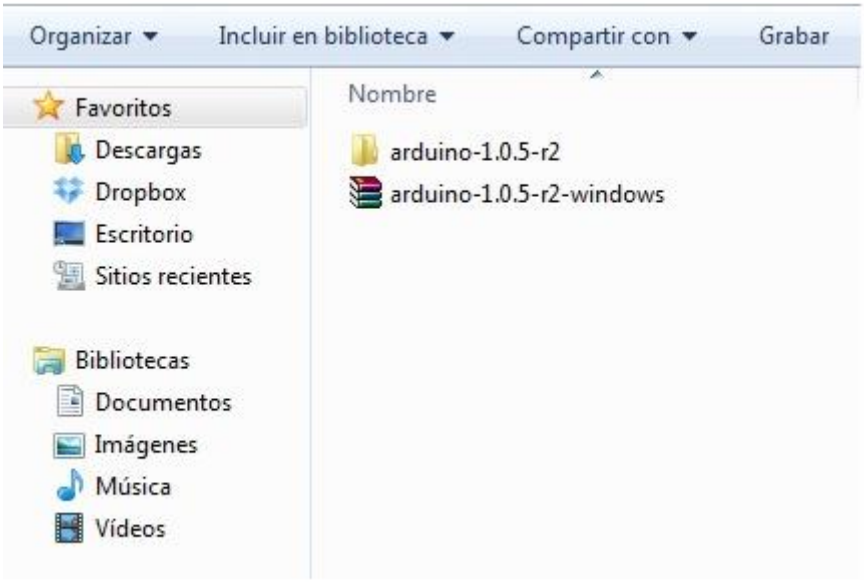


Ilustración 3

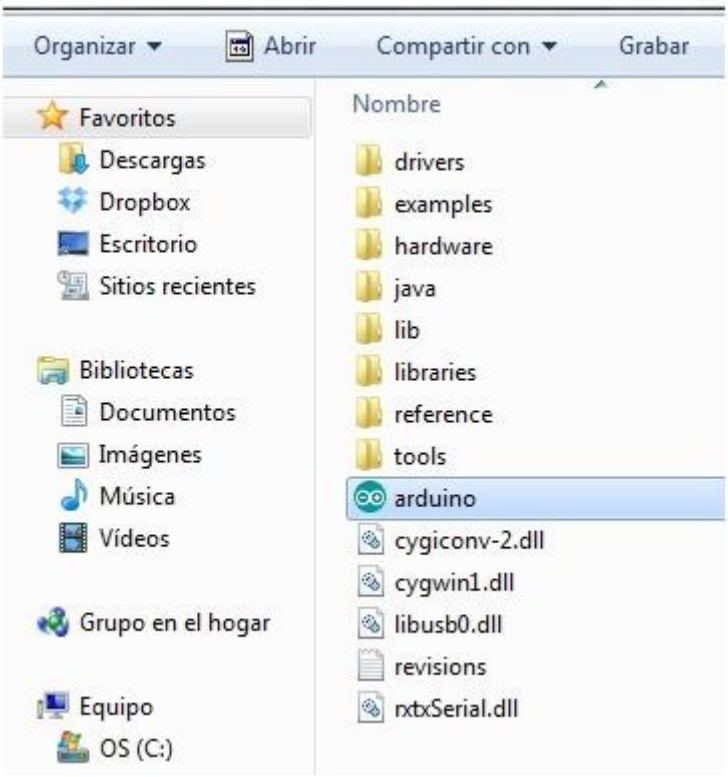


Ilustración 4

4. A continuación vamos a conectar nuestra placa Arduino al puerto USB y esperaremos a que se instalen los drivers oportunos, observamos el puerto que ha sido asignado a nuestra placa (en nuestro caso COMxx).

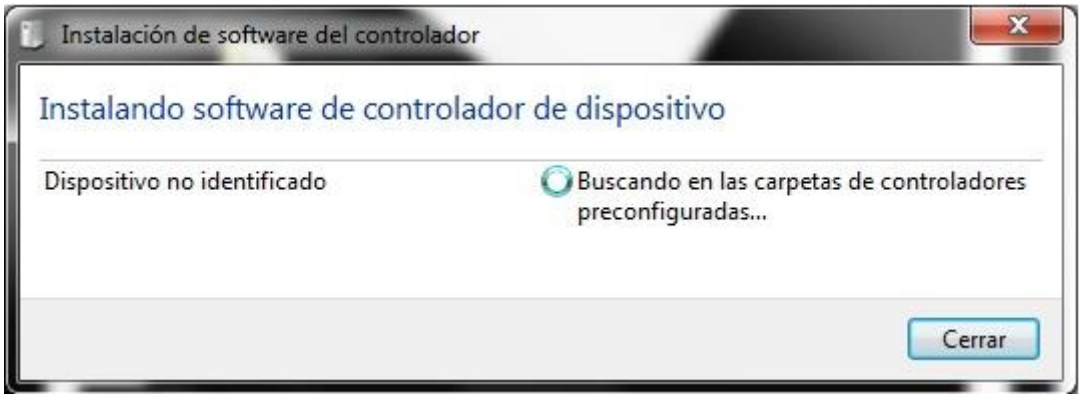


Ilustración 5

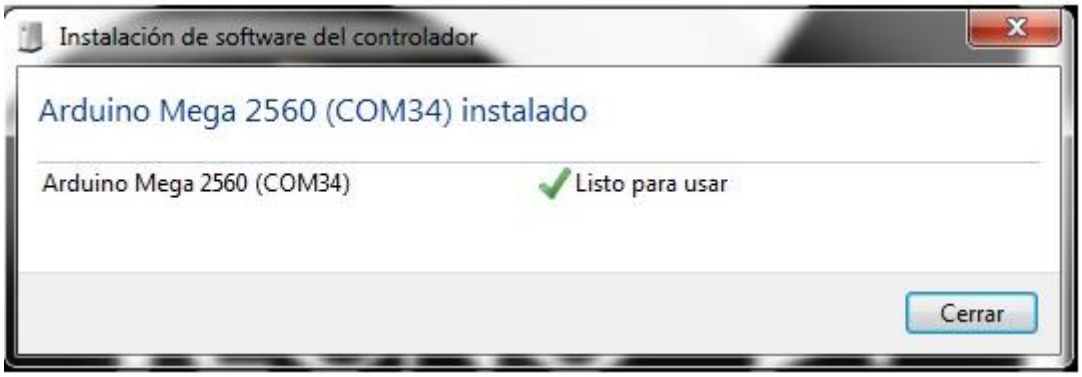


Ilustración 6

No siempre resulta tan fácil instalar los drivers, de no ser así, podemos encontrarnos con dos situaciones, dependiendo del chip de comunicación utilizado por Arduino.

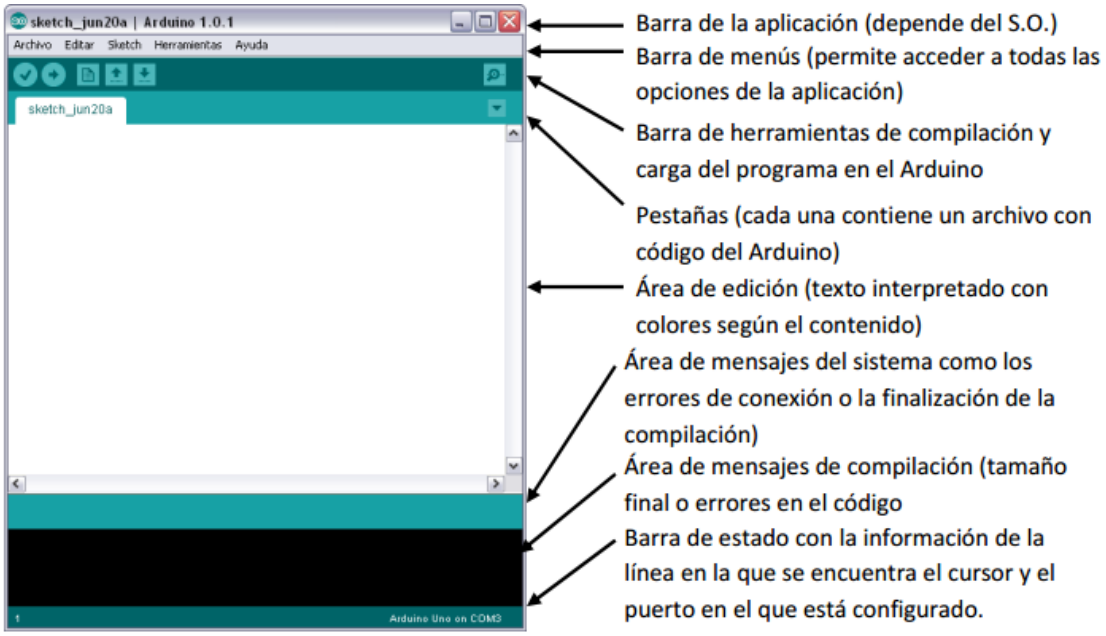
Si es un chip FT232 (típico de comunicaciones serie), Windows lo reconocerá y para instalarles habrá que ir ***Dispositivos e Impresoras*** y pinchamos sobre el dispositivo correspondiente, el cual tendrá triangulo amarillo como el descrito en la siguiente imagen. Le damos a actualizar controlador, permitiendo que busque los drivers por internet y listo.

Ilustración 7

Para el caso de que nuestra placa utilice un chip utilizado básicamente por Arduino, deberemos de hacer prácticamente lo mismo, con la diferencia de que esta vez le diremos de donde debe de tomar los drivers para dicho dispositivo, los cuales estarán localizados en la carpeta de drivers dentro de nuestra carpeta de Arduino.

5. Una vez que tenemos descargada y descomprimida la aplicación Arduino y los drivers de nuestra placa instalados, con la placa conectada al PC, abrimos la aplicación de Arduino.

3.2 El entorno de desarrollo



3.2.1 Barra de herramientas

- **Verificar**

Ilustración 8

Verifica (compilando previamente) el código para comprobar posibles errores, mostrándolos de haberles en el área de mensajes de compilación.

- **Cargar**

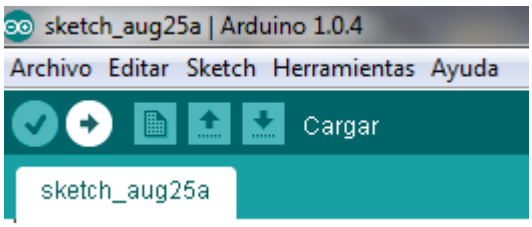


Ilustración 9

Compila y carga el programa en nuestra palca. (Ha de seleccionarse previamente la placa, puesto y tipo de programador.)

- **Nuevo**

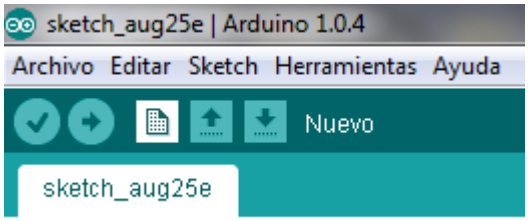


Ilustración 10

Crea un sketch nuevo. (Nuevo programa)

- **Abrir**

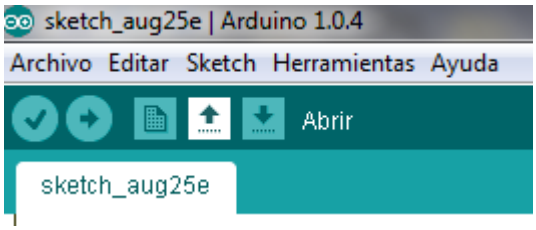


Ilustración 11

Abre un sketch en la ventana actual.

- **Guardar**

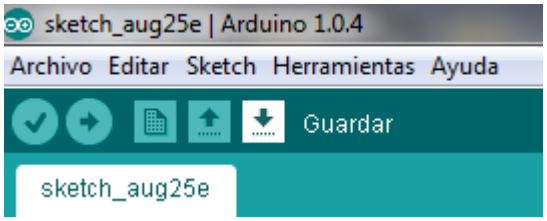


Ilustración 12

Almacena el sketch actual en el disco.

- **Monitor serial**

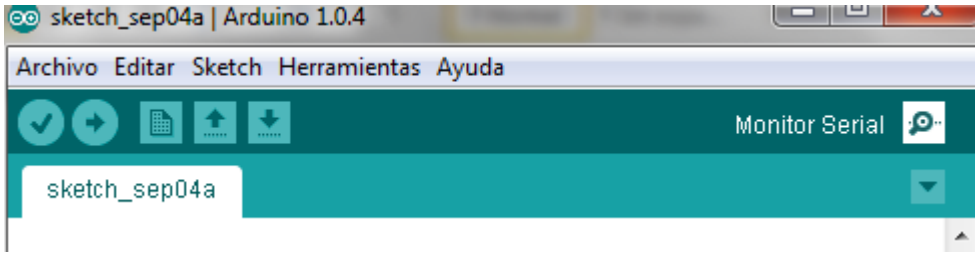


Ilustración 13

Abre la ventana del monitor de la comunicación serie.

- **Ventana**

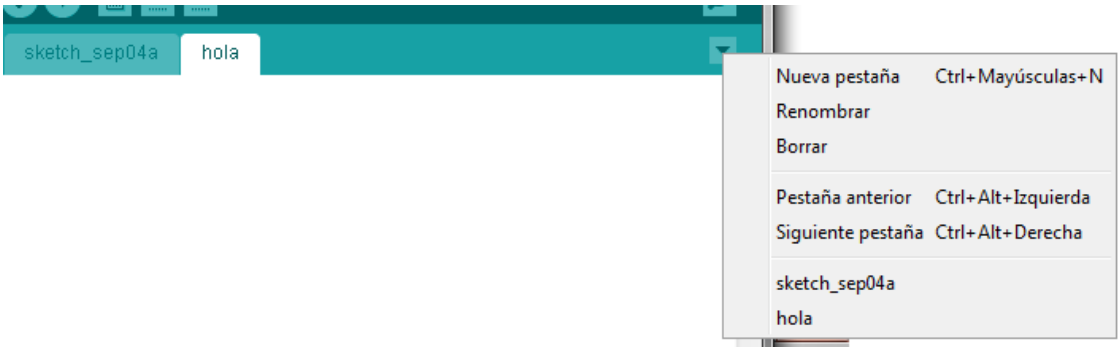


Ilustración 14

Nos permite editar y seleccionar pestañas correspondientes a un m ismo código, con la finalidad entre otras de ordenar un programa.

3.2.2 Menú Archivo

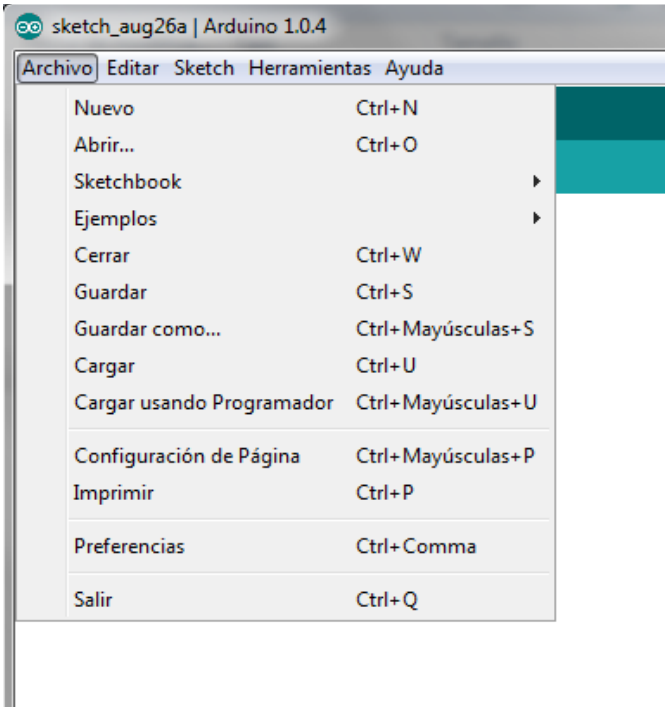
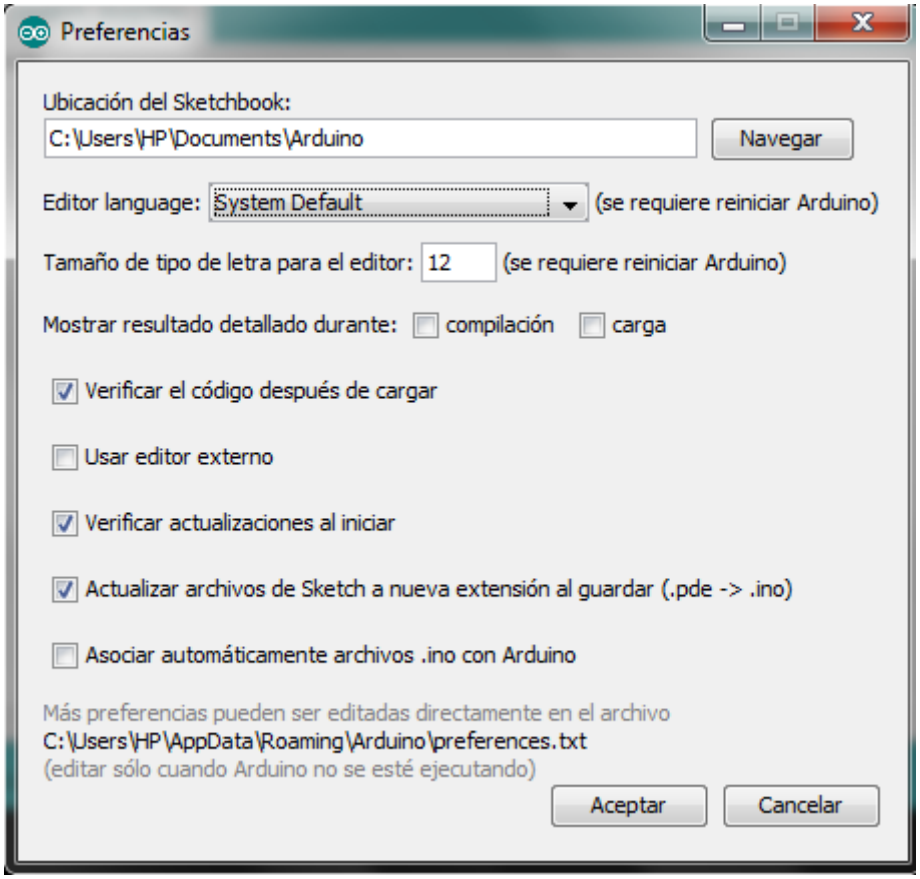


Ilustración 15

- **Nuevo**. Genera un nuevo sketch, por defecto en una ventana nueva
- **Abrir**. Abre un sketch a partir de un archivo del disco
- **Sketchbook**. Abre la carpeta “Arduino” donde se almacenan (por defecto) los sketches que se vayan editando
- **Ejemplos**. Muestra los ejemplos básicos de funcionamiento de Arduino

- **Cerrar.** Cierra la actual ventana con el sketch
- **Guardar**. Guarda el sketch actual con el nombre por defecto
- **Guardar Como**. Guarda el sketch con un nombre diferente.
- **Cargar**. Compila el sketch y lo carga en la placa Arduino que se haya seleccionado por el puerto correspondiente.
- **Cargar usando programador**. Carga el sketch usando un programador diferente al de Arduino (el programador debe estar conectado al puerto USB)
- **Configuración de página**. Configura los márgenes y la visualización de la página para imprimir.
- **Imprimir**. Imprime la página del sketch
- **Preferencias**. Permite configurar el comportamiento de la aplicación. También da acceso directo al archivo de configuración (con más características configurables)
- **Salir**. Sale de la aplicación
- Preferencias.



3.2.3 Menú editar

- **Deshacer**. Deshace la última acción de edición.
- **Rehacer**. Rehace la última acción de edición.
- **Cortar**. Corta (elimina) el texto seleccionado para ser pegado en otra posición.
- **Copiar**. Copia (sin eliminar) el texto seleccionado para ser pegado en otra posición del sketch.
- **Copiar para el foro**. Copia el texto en formato maquetado y coloreado adecuado para ser publicado en el foro de Arduino.
- **Copiar como HTML**. Copia el código, incluyendo el formato (colores, tabulados, etc.) con etiquetas HTML adecuado para ser publicado en Web.
- **Pegar**. Pega el contenido del portapapeles en la posición actual del cursor.
- Seleccionar todo. Selecciona todo el código del sketch.
- **Comentar/Descomentar**. Añade/Quita los símbolos de los comentarios ('//') en cada una de las líneas del sketch.
- **Incrementar margen**. Añade una tabulación a cada una de las líneas del texto seleccionado.
- **Reducir margen**. Quita una tabulación a cada una de las líneas del texto seleccionado.
- **Buscar**. Abre la ventana de búsqueda (contiene más opciones que el menú)

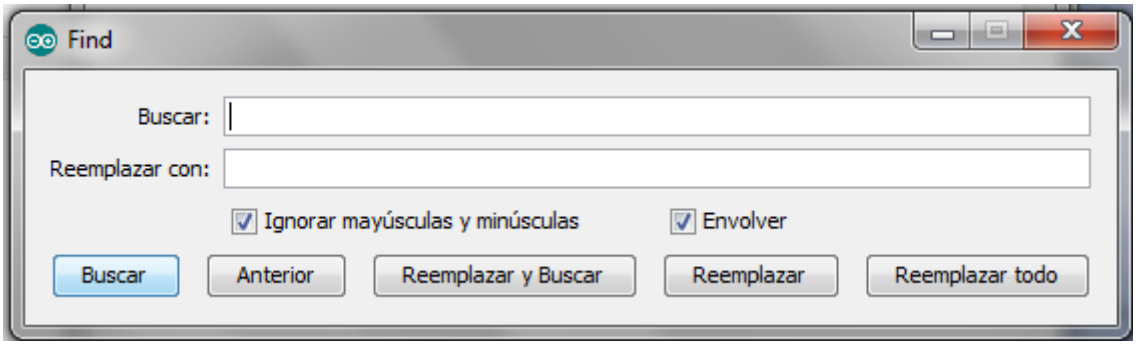


Ilustración 16

- **Buscar siguiente**. Busca la siguiente aparición de la cadena buscada a partir de la posición actual del cursor (búsqueda "hacia delante")
- **Buscar anterior**. Busca la anterior aparición de la cadena buscada a partir de la posición actual del cursor (búsqueda "hacia atrás")

- **Utilizar actual selección para buscar**. Emplea el texto seleccionado para buscar en el sketch

3.2.4 Menú Sketch

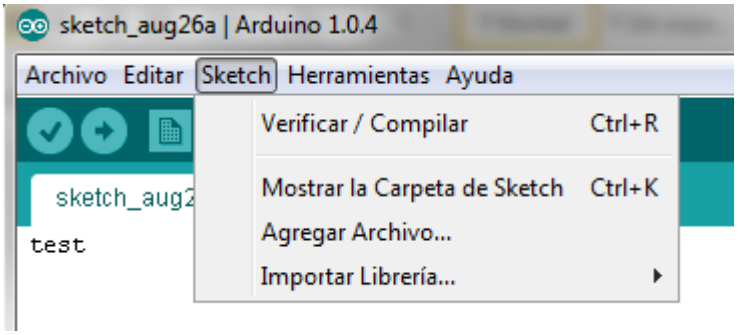


Ilustración 17

- **Verificar/compilar**. Comprueba la sintaxis del código (llaves bien cerradas, etc.) y seguidamente (en caso de ser correcta la comprobación anterior) compila el código. De haber errores aparecerán en el área de mensajes.
- **Mostrar la carpeta del sketch**. Muestra la carpeta donde está almacenado el sketch que se está editando actualmente (de gran utilidad para comprobar duplicados)
- **Agregar Archivo**. Abre un archivo del disco y lo añade una nueva pestaña.
- **Importar Librería**. Incluye la cabecera que permite importar las librerías que estén en el directorio “Libraries” de la carpeta donde se encuentra el IDE, apareciendo de la forma: #include (librería).

  Para añadir librerías que no se encuentren previamente en nuestro IDE, solo hay que descargarlas en la siguiente carpeta: Arduino(versión)/libraries

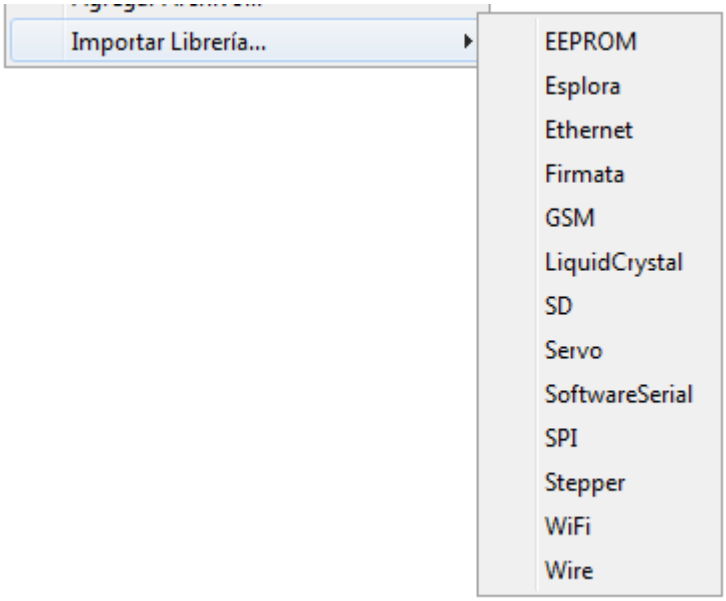


Ilustración 18

3.2.5 Menú Herramientas

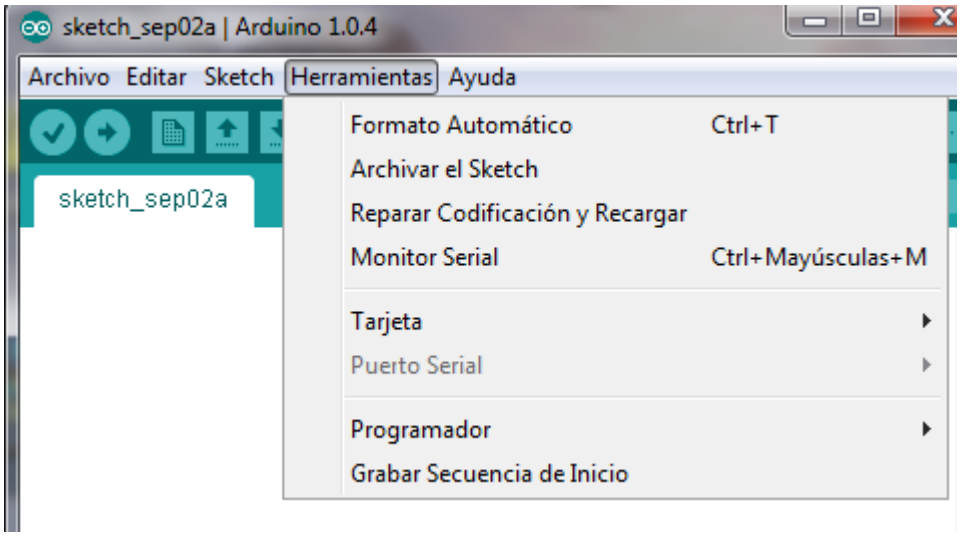
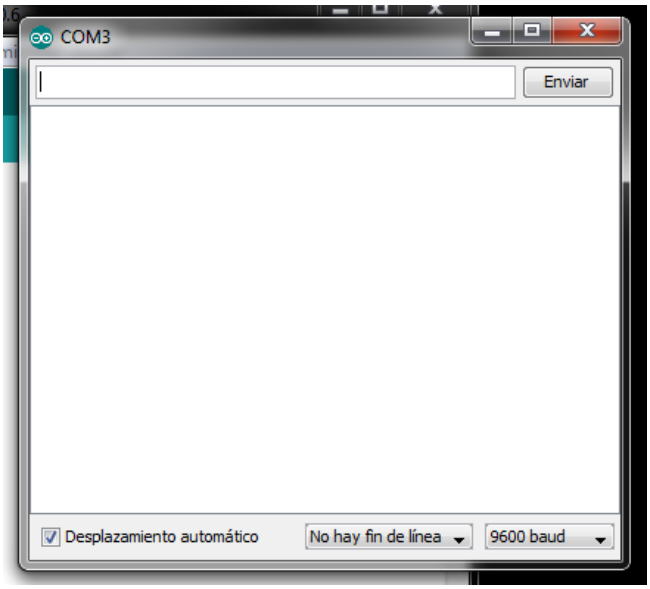


Ilustración 19

- **Formato automático**. Realiza labores de formato automático del código. Añade las tabulaciones donde son necesarias, iguala llaves y tareas similares. De gran utilidad cuando se pega código copiado de Internet o de archivos de texto.
- **Archivar el sketch**. Genera un archivo comprimido con todos los archivos de código pertenecientes al sketch.
- **Reparar codificación y recargar**. Limpia el código de caracteres no ASCII
- **Monitor Serial**. Abre la ventana del monitor serie que permite ver las transferencias de caracteres entre el Arduino y el ordenador por el puerto serie



Si los baudios no coinciden con los de nuestro sketch, aparecerán datos incoherentes.

- **Tarjeta**: permite seleccionar la tarjeta Arduino con la que trabajar
- **Serie**: permite seleccionar el puerto serie para las comunicaciones (para Windows aparecerán como COM, mientras que en Linux aparcaran como ACM o USB, dependiendo de la placa.
- **Programador**. Selecciona el cargador de código que se empleará para enviarlo al Arduino (los cargadores se encuentran en la carpeta “hardware\arduino\bootloaders” del directorio donde se encuentre el IDE
- **Grabar secuencia de inicio**. Carga la secuencia de “boot” (bootloader) en el Arduino (precisa de un cargador en el USB o de otra placa Arduino, Arduino Uno, etc)

3.2.6 Ayuda.

Acceso a los archivos de ayuda en red y la información de la versión del IDE que se está empleando.

4 Empezando a programar. Como realizar un programa de Arduino.

El lenguaje con el que se programan nuestras Arduinos es un derivado del lenguaje de programación **Wiring [14]**. Su estructura y su sintaxis es igual que la de C, por eso cualquier persona que sepa programar en C sabe programar un Arduino. De hecho todas las librerías de **C** y algunas de **C++** se pueden utilizar con Arduino.

4.1 Estructura [15]

Un programa diseñado para ejecutarse sobre un Arduino (un "sketch") siempre se compone de tres partes principales:

- **La zona global**

  Aquí será donde indicaremos a Arduino los nombres de los pines y donde crearemos aquellas variables y constantes globales que queramos que existan en el programa. Además en esta parte (al principio del todo), añadiremos librerías con funciones que queramos utilizar y que no estén incluidas en la librería estándar de Arduino.

- **La función void setup().** Delimitada por llaves de apertura y cierre {}.

Esta función se ejecuta cada vez que se inicia Arduino (incluyendo al pulsar RESET). Una de las operaciones que se realiza en **void setup()** es la de configurar de los pines que vamos a utilizar. Otra de las cosas que se suele realizar en esta parte es inicializar las comunicaciones serie.

- **La función void loop().** Delimitada por llaves de apertura y cierre {}.

La función loop se ejecuta a continuación e incluye el código que se ejecuta continuamente leyendo entradas, activando salidas, etc. Esta función permanece en ejecución en forma de bucle infinito. Esto quiere decir que se ejecuta de comienzo a fin, de forma repetida, siempre.

4.2 Que es una función.

Una función es un bloque de código que tiene un nombre y un conjunto de instrucciones que son ejecutadas cuando se llama a la función. Son funciones setup() y loop() de las que ya se ha hablado.

Las funciones de usuario pueden ser escritas para realizar tareas repetitivas y para reducir el tamaño de un programa. Las funciones se declaran asociadas a un tipo de valor. Este valor será el que devolverá la función, por ejemplo 'int' se utilizará cuando la función devuelva un dato numérico de tipo entero. Si la función no devuelve ningún valor entonces se colocará delante la palabra "void", que significa "función vacía". Después de declarar el tipo de dato que de-

vuelve la función se debe escribir el nombre de la función y entre paréntesis se escribirán, si es necesario, los parámetros que se deben pasar a la función para que se ejecute.

4.3 Mayúsculas y puntos y comas

El lenguaje Arduino es “case-sensitive”. Esto quiere decir que es totalmente diferente escribir una letra en mayúscula que en minúscula.

Todas las instrucciones (incluyendo también las declaraciones de variables) acaban con un punto y coma. Es indispensable añadir siempre este signo para no tener errores de compilación, ya que el compilador necesita localizarlo para poder detectar el final de cada instrucción escrita en el sketch.

4.4 Comentarios

Los comentarios son texto de ayuda con la finalidad de explicar el código asociado y ayudar a entenderlo y recordar su función.

Los comentarios son completamente ignorados y desechados por el compilador.

Hay dos tipos de comentarios. (Iguales que para el lenguaje C).

- Comentarios compuestos por una línea entera (o parte de ella): para ello, se añaden dos barras (**//**) al principio de la línea o parte de la línea que queramos convertir en comentario.

- Comentarios compuestos por un bloque de varias líneas seguidas: para añadirlos deberemos escribir una barra seguida de un asterisco (**/***) al principio del bloque de texto que queramos convertir en comentario, y un asterisco seguido de una barra **(*/**) al final de dicho bloque.

4.5 Variables

Cuando creamos un programa siempre se manejan datos, los cuales lo podemos almacenar como datos variables. El contenido de las variables se puede ver o cambiar en cualquier momento. Estas variables pueden ser de distintos tipos dependiendo del tipo de dato que queramos meter. No es lo mismo guardar un nombre que un número. Hay que recordar también que la memoria del ordenador es limitada, así que cuando guardamos un dato, debemos usar sólo la memoria necesaria. Por ejemplo si queremos almacenar el número 400 usaremos una variable tipo ***int*** (lo veremos más adelante) que ocupa sólo 16 bits, y no una de tipo ***long*** que ocupa 32 bits. Además ahorrar memoria es una buena costumbre.

Antes de utilizar una variable, hay que declararla, con su nombre y tipo.

4.5.1 El nombre de una variable

A las variables no se les puede dar cualquier nombre. No se pueden poner más que letras de la 'a' a la 'z' (la ñ no vale), números y el símbolo '_'. No se pueden poner signos de admiración, ni de interrogación... **El nombre de una variable puede contener números, pero su primer carácter no puede serlo.**

Ejemplos de nombres válidos:

```
numero

buFfer

a1

j10hola29

num_alumnos
```

Ejemplos de nombres no válidos:

```
1abc
nombre?
num/alumnos
```

También hay que saber que se distinguen las mayúsculas de las minúsculas, siendo estos dos ejemplos diferentes variables:

hola

Hola

4.5.2 Asignación de valores a una variable

¿Qué pasa si una variable se declara pero no se inicializa? Tendrá un valor por defecto (normalmente sin interés para nosotros) hasta que se le asigne un valor diferente en algún momento de la ejecución de nuestro sketch. La sintaxis general para asignar un nuevo valor a una variable (ya sea porque no se ha inicializado, o sobre todo porque se desea sobrescribir un valor anterior por otro nuevo), es:

nombreVariable = nuevoValor;

4.5.3 Ámbito de una variable

Que sea de un ámbito o de otro depende de en qué lugar de nuestro sketch se declare la variable:

Para que una variable sea global se ha de declarar al principio del sketch; es decir, antes (y fuera) de las secciones “void setup()” y “void loop.

Una variable global es aquella que puede ser utilizada y manipulada desde cualquier punto del sketch. Es decir, todas las instrucciones de nuestro programa, sin importar dentro de qué sección estén escritas (“void setup()”, “void loop()” u otras que puedan existir) pueden consultar y también cambiar el valor de dicha variable.

Para que una variable sea local se ha de declarar en el interior de alguna de las secciones de nuestro sketch (“void setup()” o de “void loop()” o dentro de una función, fuera de estas dos partes mencionadas).

Una variable local es aquella que solo puede ser utilizada y manipulada por las instrucciones escritas dentro de la misma sección donde se ha declarado. Este tipo de variables es útil es sketches largos y complejos para asegurarse de que solo una sección tiene acceso a sus propias variables, ya que esto evita posibles errores cuando una sección modifica inadvertidamente variables utilizadas por otra sección.

4.5.4 Tipos de variables.[16]

- Boolean.

  Las variables de este tipo solo pueden tener dos valores: cierto o falso.

  Para asignar explícitamente a una variable de tipo “boolean” el valor de cierto, se puede utilizar la palabra especial “true” (sin comillas) o bien el valor “1” (sin comillas), y para asignarle el valor de falso se puede utilizar la palabra especial “false” (sin comillas) o bien el valor “0” (sin comillas).

  Una variable booleana con un valor cualquiera diferente de 0 ya se interpreta que tiene un valor cierto: no tiene por qué ser el valor 1 concretamente.

  Ejemplo:

  boolean mivariable=true;

  boolean mivariable=1;

- **Enteros**.

  Los enteros se usan para representar números enteros, y pueden ser de varios tipos, en función del espacio de memoria que ocupen.

  1. *Tipo int*

  El valor que puede tener una variable de este tipo es un número entero entre -32768 (-215) y 32767 (215-1), ya que utiliza dos bytes (16 bits) de memoria para almacenarse. Esto es así para todas las placas Arduino excepto para la Due, la cuel reserva 32 bits.

  Ejemplo:

  int ledPin = 13;

  sintaxis:

  int var = val;

  var = nombre de la variable de tipo int

  val = valor asisgando a la variable de tipo int.

  2. *Tipo short*

Es un entero que ocupa la mitad de un int, pero solo para el caso del Arduino Due, en el cual ocupa 4bytes como int y 2 bytes como short, para el resto de placas un entero int y short, ocuparan y podrán tomar los mismos valores, 2 bytes.

3. *Tipo long*

Los enteros de tipo long, ocuparan 4 bytes, números entre -2147483648 a 2147483647. (los de tipo long y int para el Arduino Due, son equivalentes).

4. *Unsigned long*

Tomará valores enteros, los cuales ocupen 4 bytes, pero solo podrán ser valores positivos, yendo de 0 a 4.294.967.295.

5. *unsigned int*

Estos valores enteros, ocupa5ran 2 bytes, excepto para el Arduino Due que será de 4 bytes, pudiendo tomar solo valores positivos.

- **Tipo byte**

  El valor que puede tener una variable de este tipo es siempre un número entero entre 0 y 255. Al igual que las variables de tipo “char”, las de tipo “byte” utilizan un byte (8 bits) para almacenar su valor y, por tanto, tienen el mismo número de combinaciones numéricas posibles diferentes (256).

- **Tipo char**

  El valor que puede tener una variable de este tipo es siempre un solo carácter (una letra, un dígito, un signo de puntuación...). Un byte.

  Para asignar explícitamente a una variable de tipo “char” un determinado valor (es decir, un carácter), debemos de escribirlo ese carácter, entre comillas.

  Los caracteres son almacenados de manera numérica en la memoria, tomando valores de -128 a 127, siendo válida su representación, o en su forma de carácter, o en su forma numérica, como podemos ver en el ejemplo, en el cual el carácter A corresponde en ASCII **[17]** al número 65.

  También podemos almacenar el carácter en forma de número,

  Ejemplo:

  ```
  char miCaracter = 'A';

  char miCaracter = 65;
  ```

- **El tipo word**

  La variable de tipo word, ocupa como un int, 2 bytes, salvo para el Arduino due (4byte), con la diferencia que solo puede tomar valores positivos, de 0 a 65535.

- **Tipo float**

  El valor que puede tener una variable de este tipo es un número decimal. Los valores "float" posibles pueden ir desde el número -3,4028235·1038 hasta el número 3,4028235·1038 (4 bytes de memoria). Debido a su grandísimo rango de valores posibles, los números decimales son usados frecuentemente para aproximar valores analógicos continuos. No obstante, solo tienen 6 o 7 dígitos en total de precisión. Es decir, los valores "float" no son exactos, y pueden producir resultados inesperados, como por ejemplo que, 6.0/3.0 no dé exactamente 2.0.

  Otro inconveniente de los valores de tipo "float" es que el cálculo matemático con ellos es mucho más lento que con valores enteros, por lo que debería evitarse el uso de valores "float" en partes de nuestro sketch que necesiten ejecutarse a gran velocidad.

  Los números decimales se han de escribir en nuestro sketch utilizando la m notación anglosajona (es decir, utilizando el punto decimal en vez de la coma). Si lo deseamos, también se puede utilizar la notación científica (es decir, el número 0,0234 – equivalente a 2,34·10-2 – lo podríamos escribir por ejemplo como 2.34e-2)

- **Array**

  Un array (también llamado "vector") es una colección de variables de un tipo concreto que tienen todos el mismo y único nombre, pero que pueden distinguirse entre sí por un número a modo de índice. Es decir: en vez de tener diferentes variables –por ejemplo de tipo "char" – cada una independiente de las demás (varChar1, varChar2,varChar3...) podemos tener un único array que las agrupe todas bajo un mismo nombre (por ejemplo, varChar), y que permita que cada variable pueda manipularse por separado gracias a que dentro del array cada una está identificada mediante un índice numérico, escrito entre corchetes (varChar[0], varChar[1], varChar[2]...). Los arrays sirven para ganar claridad y simplicidad en el código, además de facilitar la programación.

  Un array se puede declarar de las siguientes maneras:

  int varInt[6]; Declara un array de 6 elementos (es decir, variables individuales) sin inicializar ninguno.

  int varInt[] = {2,5,6,7}; Declara un array sin especificar el número de elementos. No obstante, se asignan entre llaves, separados por comas, los valores directamente, por lo que el compilador es capaz de deducir el número de elementos total del array (en este ejemplo, cuatro).

  int varInt[8] = {2,5,6,7}; Declara un array de 8 elementos e inicializa algunos de ellos (los cuatro primeros), dejando el resto sin inicializar. Lógicamente, si se inicializaran más elementos que lo que permite el tamaño del array (por ejemplo, si se asignan 9 valores a un array de 8 elementos) se produciría un error.

```
char varChar[6] = “hola”;

char varChar[6] = {'h','o','l','a'};

char varChar[] =”hola”;
```

La primera forma declara e inicializa un array de seis elementos de tipo “char”. Como los arrays de tipo “char” son en realidad cadenas de caracteres (es decir: palabras o frases, “strings” en inglés) tienen la particularidad de poder inicializarse tal como se muestra en la primera forma: indicando directamente la palabra o frase escrita entre comillas dobles. Pero también se pueden declarar como un array “estándar”, que es como muestra la segunda forma. Observar en este caso la diferencia de comillas: un carácter individual siempre se especifica entre comillas simples, y el valor de una cadena siempre se especifica entre comillas dobles. También es posible, tal como muestra la tercera forma, declarar una cadena sin necesidad de especificar su tamaño (ya que el compilador lo puede deducir a partir del número de elementos – es decir, caracteres– inicializados).

Hay que tener en cuenta que el primer valor del array tiene el índice 0 y por tanto, el último valor tendrá un índice igual al número de elementos del array menos uno.

Cuidado con esto, porque asignar valores más allá del número de elementos declarados del array es un error. En concreto, si por ejemplo tenemos un array de 2 elementos de tipo entero (es decir, declarado así: int varInt[2];), para asignar un nuevo valor (por ejemplo, 27) a su primer elemento deberíamos escribir:

```
varInt[0] = 27;
```

y para asignar el segundo valor (por ejemplo, 17), escribiríamos:

```
varInt[1] = 17;
```

Pero si asignáramos además un tercer valor así,

```
varInt[2] = 53;
```

Cometeríamos un error porque estaríamos sobrepasando el final previsto del array (y por tanto, utilizando una zona de la memoria no reservada, con resultados imprevisibles).

En el caso concreto de los arrays de caracteres (las “**cadenas**” o “**strings**”), hay que tener en cuenta una particularidad muy importante: este tipo de arrays siempre deben ser declarados con un número de elementos una unidad mayor que el número máximo de caracteres que preveamos guardar. Es decir, si se va a almacenar la palabra “hola” (de cuatro letras), el array deberá ser declarado como mínimo de 5 elementos. Esto es así porque este último elemento siempre se utiliza para almacenar automáticamente un carácter especial (el carácter “nulo”, con código ASCII 0), que sirve para marcar el final de la cadena. Esta marca es necesaria para que el compilador sepa que la cadena ya terminó y no intente seguir leyendo más posiciones.

Finalmente, comentar que a menudo es conveniente, cuando se trabaja con grandes cantidades de texto (por ejemplo, en un proyecto con pantallas LCD), utilizar arrays de strings. Para declarar un array de esta clase, se utiliza el tipo de datos especial "char*" (notar el asterisco final). Un ejemplo de declaración e inicialización de este tipo sería: char* varCadenas[]={"Cad0","Cad1","Cad2","Cad3"};.

En realidad, el asterisco en la declaración anterior indica que en realidad estamos declarando un array de "punteros", ya que para el lenguaje Arduino, las cadenas son punteros. Los "punteros" son unos elementos del lenguaje Arduino (provenientes del lenguaje C en el que está basado) muy potentes pero a la vez ciertamente complejos.

4.6 Constantes

Es posible declarar una variable de tal forma que consigamos que su valor (del tipo que sea) permanezca siempre inalterado. Es decir, que su valor no se pueda modificar nunca porque esté marcado como de "solo lectura". De hecho, a este tipo de variables ya no se les llama así por motivos obvios, sino "constantes". Las constantes se pueden utilizar como cualquier variable de su mismo tipo, pero si se intenta cambiar su valor, el compilador lanzará un error.

Para convertir una variable (sea global o local) en constante, lo único que hay que hacer es preceder la declaración de esa variable con la palabra clave const. Por ejemplo, para convertir en constante una variable llamada "sensor" de tipo "byte", simplemente se ha de declarar así: const byte sensor;

Existe otra manera de declarar constantes en el lenguaje Arduino, que es utilizando la directiva especial **#define** (heredada del lenguaje C).

Ejemplo:

#define numeroPi = 3.1416;

4.7 Funciones (instrucciones) Principales

La plataforma Arduino tiene un enorme catálogo de bibliotecas de funciones, algunas de las cuales están directamente incluidas en el entorno de desarrollo. Las que no están incluidas en él se pueden añadir con la expresión:

```
#include <biblioteca.h>
```

Vamos a explicar ciertos conceptos sobre estas funciones (instrccuiones), y las más importantes o estándares que proporciona el IDE de Arduino.

4.7.1 Parámetros de una instrucción

Ya se habrá observado que al final del nombre de cada instrucción utilizada (Serial.begin(), Serial.println(), etc.. siempre aparecen unos paréntesis. Estos paréntesis pueden estar vacíos pero también pueden incluir en su interior un número/letra/palabra, o dos, o tres, etc. (si hay más de un valor han de ir separados por comas). Cada uno de estos valores es lo que se deno-

mina un “parámetro”, y el número de ellos y su tipo (que no tiene por qué ser el mismo para todos) dependerá de cada instrucción en particular.

Los parámetros sirven para modificar el comportamiento de la instrucción en algún aspecto. Es decir: las instrucciones que no tienen parámetros hacen una sola función (su tarea encomendada) : no hay posibilidad de modificación porque siempre harán lo mismo de la misma manera. Cuando una instrucción, en cambio, tiene uno o más parámetros, también hará su función preestablecida, pero la manera concreta vendrá dada por el valor de cada uno de sus parámetros, los cuales modificarán alguna característica concreta de esa acción a realizar.

4.7.2 Valor de retorno de una instrcucción

Las instrucciones, además de recibir parámetros de entrada (si lo hacen), y aparte de hacer la tarea que tienen que hacer, normalmente también devuelven un valor de salida (o “de retorno”). Un valor de salida es un dato que podemos obtener en nuestro sketch como resultado “tangible” de la ejecución de la instrucción. El significado de ese valor devuelto dependerá de cada instrucción concreta: algunos son de control (indicando si la instrucción se ha ejecutado bien o mal), otros son resultados numéricos obtenidos tras la ejecución de algún cálculo matemático, etc.

4.7.3 La comunicación serial con la placa Arduino [18]

Ya se ha comentado anteriormente que el microcontrolador de nuestra placa dispone de un receptor/transmisor serie de tipo TTL-UART que permite comunicar la placa Arduino con otros dispositivos (normalmente, nuestro computador)

El canal físico de comunicación en estos casos suele ser el cable USB, pero también pueden ser los pines digitales 0 (RX) y 1 (TX) de la placa. Si se usan estos dos pines para comunicar la placa con un dispositivo externo, tendremos que conectar concretamente el pin TX de la placa con el pin RX del dispositivo, el RX de la placa con el TX del dispositivo y compartir la tierra de la placa con la tierra del dispositivo. Hay que tener en cuenta que si se utilizan estos dos pines para la comunicación serie, no podrán ser usados como entradas/salidas digitales estándar.

- **Serial.begin()**:

  Abre el canal serie para que pueda empezar la comunicación por él. Por tanto, su ejecución es imprescindible antes de realizar cualquier transmisión por dicho canal. Por eso normalmente se suele escribir dentro de la sección “void setup()”. Además, mediante su único parámetro –de tipo “long” y obligatorio– , especifica la velocidad en bits/s a la que se producirá la transferencia serie de los datos. Para la comunicación con un computador, se suele utilizar el valor de 9600, pero se puede especificar cualquier otra velocidad. Lo que sí es importante es que el valor escrito como parámetro coincida con el que se especifique en el desplegable que aparece en el “Serial Monitor” del IDE de Arduino, o si no la comunicación no estará bien sincronizada y se mostrarán símbolos sin sentido. Esta instrucción no tiene valor de retorno.

- **Serial.end(),**

No tiene ningún argumento ni devuelve nada, y que se encarga de cerrar el canal serie; de esta manera, la comunicación serie se deshabilita y los pines RX y TX vuelven a estar disponibles para la entrada/salida general. Para reabrir el canal serie otra vez, se debería usar de nuevo Serial.begin().

Instrucciones para enviar datos:

- **Serial.print():**

  Envía a través del canal serie un dato (especificado como parámetro) desde el microcontrolador hacia el exterior. Ese dato puede ser de cualquier tipo: carácter, cadena, número entero, número decimal (por defecto de dos decimales), etc. Si el dato se especifica explícitamente (en vez de a través de una variable), hay que recordar que los caracteres se han de escribir entre comillas simples y las cadenas entre comillas dobles.

  En el caso de que el dato a enviar sea entero, se puede especificar un segundo parámetro opcional que puede valer alguna constante predefinida de las siguientes: BIN, HEX o DEC. En el primer caso se enviará la representación binaria del número, en el segundo, la representación hexadecimal, y en el tercero, la representación decimal (la usada por defecto)

- Serial.print(78, BIN) imprime "1001110"

  Serial.print(78, OCT) imprime "116"
  Serial.print(78, DEC) imprime "78"
  Serial.print(78, HEX) imprime "4E"

  En el caso de que el dato a enviar sea decimal, se puede especificar además un segundo parámetro opcional que indique el número de decimales que se desea utilizar (por defecto son dos).

Serial.print(1.23456, 4) imprime "1.2346"

- **Serial.flush()**

  A veces ocurre que no todos los datos enviados por Seral.print, llegan. Es esos casos podemos utilizar a continuación de Serial.print(); la instrucción Serial,flush(), la cual no tiene ningún parámetro ni devuelve ningún dato, y nos servirá, para que la transmisión de los datos sea completa, ya que dicha instrucción espera hasta que todos los datos hayan sido enviados, antes de continuar la ejecución del sketch.

- **Serial.println():**

  Hace exactamente lo mismo que Serial.print(), pero además, añade automáticamente al final de los datos enviados dos caracteres extra: el de retorno de carro (código ASCII nº 13) y el de nueva línea (código ASCII noº 10). La consecuencia es que al final de la ejecución de Serial.println() se efectúa un salto de línea. Tiene los mismos parámetros y los mismos valores de retorno que Serial.print()

Los siguientes dos ejemplos, hacen lo mismo, siendo el carácter '/r' el de retorno de carro y '/n' el de linea nueva.

```
Serial.print("hola\r\n");

Serial.println("hola");
```

- **Serial.write():**

  Envía a través del canal serie un dato (especificado como parámetro) desde el microcontrolador hacia el exterior. Pero a diferencia de Serial.print(), el dato a enviar solo puede ocupar un byte. Por lo tanto, ha de ser básicamente de tipo "char" o "byte".

Tambien puede enviar cadenas de caracteres.

Instrucciones para la lectura de datos:

- **Serial.available():**

  Devuelve el número de bytes –caracteres– disponibles para ser leídos que provienen del exterior a través del canal serie (vía USB o vía pines TX/RX). Estos bytes ya han llegado al microcontrolador y permanecen almacenados temporalmente en una pequeña memoria (buffer)de 64 bytes que tiene el chip TTL-UART.

  Si no hay bytes para leer, esta instrucción devolverá 0.

- **Serial.read():**

  Devuelve el primer byte aún no leído de los que estén almacenados en el buffer de entrada del chip TTL-UART. Al hacerlo, lo elimina de ese buffer. Para leer el siguiente byte, se ha de volver a ejecutar Serial.read(). Y hacer así hasta que se hayan leído todos. Cuando no haya más bytes disponibles, Serial.read() devolverá -1. No tiene parámetros.

  Existen otras instrucciones además de Serial.read(), para leer datos del buffer de entrada del chip TTL-UART de formas más específicas, las cuales se explicaran a continuación, las más relevantes:

- **Serial.peek():**

  Devuelve el primer byte aún no leído de los que estén almacenados en el buffer de entrada. No obstante, a diferencia de Serial.read(), ese byte leído no se borra del buffer, con lo que las próximas veces que se ejecute Serial.peek() –o una vez Serial.read()– se volverá a leer el mismo byte. Si no hay bytes disponibles para leer, Serial.peek() devolverá -1. Esta instrucción no tiene parámetros.

- **Serial.find()**:

  Lee datos del buffer de entrada (eliminándolos de allí) hasta que se encuentre la cadena de caracteres (o un carácter individual) especificada como parámetro, o bien se ha-

yan leído todos los datos actualmente en el buffer. La instrucción devuelve “true” si se encuentra la cadena o “false” si no.

A parte de estas hay otras instrucciones como:

- **Serial.findUntil():**

http://arduino.cc/en/Serial/FindUntil

- **Serial.readBytes():**

http://arduino.cc/en/Serial/ReadBytes

- **Serial.readBytesUntil():**

http://arduino.cc/en/Serial/ReadBytesUntil

- **Serial.setTimeout():**

http://arduino.cc/en/Serial/setTimeout

- **Serial.parseFloat():**

http://arduino.cc/en/Serial/ParseFloat

- **Serial.parseInt():**

http://arduino.cc/en/Reference/ParseInt

4.7.4 Funciones de entradas y salidas digitales

Las funciones que nos ofrece el lenguaje Arduino para trabajar con entradas y salidas digitales son:

- **pinMode():**

  Configura un pin digital (cuyo número se ha de especificar como primer parámetro) como entrada o como salida de corriente, según si el valor de su segundo parámetro es la constante predefinida INPUT o bien OUTPUT, respectivamente. Esta función es necesaria porque los pines digitales a priori pueden actuar como entrada o salida, pero en nuestro sketch hay que definir previamente si queremos que actúen de una forma u de otra. Es por ello que esta función se suele escribir dentro de “setup()”. No tiene valor de retorno.

  Si el pin digital se quiere usar como entrada, es posible activar una resistencia “pull-up” de 20 KΩ que todo pin digital incorpora. Para ello, se ha de utilizar la constante predefinida INPUT_PULLUP en vez de INPUT, ya que la constante INPUT desactiva explícitamente estas resistencias “pull-ups” internas. Recordemos que si un pin de entrada tiene su resistencia “pull-up” interna desactivada, en el momento que no esté conectado a nada puede recibir ruido eléctrico del entorno o de algún pin cercano y provocar así inconsistencias en los valores de entrada obtenidos (ya que estos cambia-

rán aleatoriamente en cualquier momento). Esto hace que por lo general sea recomendable activar la resistencia “pull-up”.

- **digitalWrite():**

  Envía un valor ALTO (HIGH) o BAJO (LOW) a un pin digital; es decir, tan solo es capaz de enviar dos valores posibles. Por eso, de hecho, hablamos de salida “digital”. El pin al que se le envía la señal se especifica como primer parámetro (escribiendo su número) y el valor concreto de esta señal se especifica como segundo parámetro (escribiendo la constante predefinida HIGH o bien la constante predefinida LOW, ambas de tipo “int”).

  Si el pin especificado en digitalWrite() está configurado como salida, la constante HIGH equivale a una señal de salida de hasta 40 mA y de 5 V (o bien 3,3 V en las placas que trabajen a ese voltaje) y la constante LOW equivale a una señal de salida de 0 V.

- **digitalRead():**

  Devuelve el valor leído del pin digital (configurado como entrada mediante pinMode()) cuyo número se haya especificado como parámetro. Este valor de retorno es de tipo “int” y puede tener dos únicos valores (por eso, de hecho hablamos de entrada digital): la constante HIGH (1) o LOW (0).

Además de las anteriores, otra función no tan usada pero que puede ser útil:

- **pulseIn():**

  Pausa la ejecución del sketch y se espera a recibir en el pin de entrada especificado como primer parámetro la próxima señal de tipo HIGH o LOW (según lo que se haya indicado como segundo parámetro). Una vez recibida esa señal, empieza a contar los microsegundos que esta dura hasta cambiar su estado otra vez, y devuelve finalmente un valor –de tipo “long”– correspondiente a la duración en microsegundos de ese pulso de señal. De forma opcional, se puede especificar un tercer parámetro –de tipo “unsigned long”– que representa el tiempo máximo de espera en microsegundos: si la señal esperada no se produce una vez superado este tiempo de espera, la función devolverá 0 y continuará la ejecución del sketch. Si este tiempo de espera no se especifica, el valor por defecto es de un segundo. En la documentación oficial recomiendan usar esta función para rangos de valores de retorno de entre 10 microsegundos y 3 minutos, ya que para pulsos más largos la precisión puede tener errores.

4.7.5 Entradas y salidas analógicas

- **analogWrite():**

  Envía un valor de tipo “byte” (especificado como segundo parámetro) que representa una señal PWM, a un pin digital configurado como OUTPUT (especificado como primer parámetro). No todos los pines digitales pueden generar señales PWM: en la placa Arduino UNO por ejemplo solo son los pines 3, 5, 6, 9, 10 y 11 (están marcados en la placa). Cada vez que se ejecute esta función se regenerará la señal. Esta función no tiene valor de retorno.

Es importante recalcar que esta función no tiene nada que ver con los pines analógicos A0, A1, etc., ya que estos solo funcionan como pines analógicos de entrada (mediante el uso de la función analogRead()) pero no de salida. Las salidas analógicas se han de generar utilizando solamente los pines digitales PWM.

- **analogRead():**

  Devuelve el valor leído del pin de entrada analógico cuyo número (0, 1, 2...) se ha especificado como parámetro. Este valor se obtiene mapeando proporcionalmente la entrada analógica obtenida (que debe oscilar entre 0 y un voltaje llamado voltaje de referencia, el cual por defecto es 5 V) a un valor entero entre 0 y 1023. Esto implica que la resolución de lectura es de 5V/1024, es decir, de 0,049 V.

  Si quisiéramos tener más resolución en la lectura, podríamos hacerlo cambiando el voltaje de referencia, el cual como hemos visto es de 5V por defecto. Para dicho fin utilizaríamos la función analogReference()

  Como los pines analógicos por defecto solamente funcionan como entradas de señales analógicas, no es necesario utilizar previamente la función pinMode() con ellos. Estos pines también incorporan toda la funcionalidad de un pin de entrada/salida digital estándar (incluyendo las resistencias “pull-up”), por lo que si se les declara como tal, pueden utilizarse como entradas o salidas digitales, simplemente identificándolos con un número correlativo más allá del pin 13, que es el último pin digital. Es decir, el pin “A0” sería el número 14, el “A1” sería el 15, etc. Por ejemplo, si quisiéramos que el pin analógico “A3” funcionara como salida digital y además enviara un valor BAJO, escribríamos primero pinMode(17,OUTPUT); y luego digitalWrite(17,LOW);

  Hay que saber que el convertidor analógico/digital tarda alrededor de 100 microsegundos (0,0001s) en procesar la conversión y obtener el valor digital, por lo que el ritmo máximo de lectura en los pines analógicos es de 10000 veces por segundo. Esto hay que tenerlo en cuenta en nuestros sketches.

4.7.6 Cambiar el voltaje de referencia de las lecturas analógicas

- **analogReference():**

  Configura el voltaje de referencia usado para la conversión interna de valores analógicos en digitales. Dispone de un único parámetro, que en la placa Arduino UNO puede tener los siguientes valores: la constante predefinida DEFAULT (que equivale a establecer el voltaje de referencia es 5 V –o 3,3 V en las placas que trabajen a esa tensión–, el cual es el valor por defecto), o la constante predefinida INTERNAL (donde el voltaje de referencia entonces es de 1,1 V) o la constante predefinida EXTERNAL (donde el voltaje de referencia entonces es el voltaje que se aplique al pin AREF –“Analogue REFerence”–). Es muy importante ejecutar siempre esta función antes de cualquier lectura realizada con analogRead() para evitar daños en la placa. Esta función no tiene valor de retorno.

- Si usamos la opción de utilizar una fuente de alimentación externa para establecer el voltaje de referencia, deberemos conectar el borne positivo de esa nfuente al pin-

hembra AREF de la placa y su borne negativo a la tierra común. Para no dañar la placa, es muy importante que el voltaje de referencia externo aportado al pin AREF no sea nunca menor de 0 V (es decir, invertido de polaridad) ni mayor de 5 V.

4.7.7 Instrucciones de gestión de tiempo

- **millis():**

  Devuelve el número de milisegundos (ms) desde que la placa Arduino empezó a ejecutar el sketch actual. Este número se reseteará a cero aproximadamente después de 50 días (cuando su valor supere el máximo permitido por su tipo, que es "unsigned long"). No tiene parámetros.

- **micros():**

  Devuelve el número de microsegundos (µs) desde que la placa Arduino empezó a ejecutar el sketch actual. Este número –de tipo "unsigned long"– se reseteará a cero aproximadamente después de 70 minutos. Esta instrucción tiene una resolución de 4 µs (es decir, que el valor retornado es siempre un múltiplo de cuatro). Recordar que 1000 µs es un milisegundo y por tanto, 1000000 µs es un segundo. No tiene parámetros.

- **delay():**

  Pausa el sketch durante la cantidad de milisegundos especificados como parámetro –de tipo "unsigned long"–. No tiene valor de retorno.

- **delayMicroseconds():**

  Pausa el sketch durante la cantidad de microsegundos especificados como parámetro –de tipo "unsigned long"– . Actualmente el máximo valor que se puede utilizar con precisión es de 16383. Para esperas mayores que esta, se recomienda usar la instrucción delay(). El mínimo valor que se puede utilizar con precisión es de 3 µs. No tiene valor de retorno.

4.7.8 INSTRUCCIONES MATEMÁTICAS, TRIGONOMÉTRICAS Y DE PSEUDOALEATORIEDAD

Instrucciones matemáticas:

- **abs():**

  Devuelve el valor absoluto de un número pasado por parámetro (el cual puede ser tanto entero como decimal). Es decir, si ese número es positivo (o 0), lo devuelve sin alterar su valor; si es negativo, lo devuelve "convertido en positivo". Por ejemplo, 3 es el valor absoluto tanto de 3 como de -3.

- **min():**

Devuelve el mínimo de dos números pasados por parámetros (los cuales pueden ser tanto enteros como decimales).

- **max():**

  Devuelve el máximo de dos números pasados por parámetros (los cuales pueden ser tanto enteros como decimales).

- **constrain():**

  Recalcula el valor pasado como primer parámetro (llamémosle "x") dependiendo de si está dentro o fuera del rango delimitado por los valores pasados como segundo y tercer parámetro (llamémoslos "a" y "b" respectivamente, donde "a" siempre ha de ser menor que "b"). Los tres parámetros pueden ser tanto enteros como decimales. En otras palabras:

  Si "x" está entre "a" y "b", constrain() devolverá "x" sin modificar.

  Si "x" es menor que "a", constrain() devolverá "a"

  Si "x" es mayor que "b", constrain() devolverá "b"

- **map():**

  Modifica un valor,(primer parámetro el cual inicialmente está dentro de un rango (delimitado con su mínimo –segundo parámetro– y su máximo –tercer parámetro–) para que esté dentro de otro rango (con otro mínimo –cuarto parámetro– y otro máximo –quinto parámetro–) de forma que la transformación del valor sea lo más proporcional posible. Esto es lo que se llama "mapear" un valor: los mínimos y los máximos del rango cambian y por tanto los valores intermedios se adecúan a ese cambio. Todos los parámetros son de tipo "long", por lo que se admiten también números enteros negativos, pero no números decimales: si aparece alguno en los cálculos internos de la instrucción, éste será truncado. El valor devuelto por esta instrucción es precisamente el valor mapeado.

  Ejemplo:

  Tenemos un potenciómetro conectado en la entrada analógica, la cual admite valores de entre 0 y 5V (utilizaremos ente 0 y 5000, para obtenerlo en mV), pero como hemos visto, la lectura de la entrada analógica nos devuelve valores entre 0 y 1023, si quisiéramos saber la tensión, podremos hacerlo mediante la instrucción map();

```
int tension_entrada;

void setup() {}

void loop()
{
```

```
  int val = analogRead(0);
  tension_entrada = map(val, 0, 1023, 0, 5000);

Serial.println(tension_entrada);
}
```

- **pow():**

  Devuelve el valor resultante de elevar el número pasado como primer parámetro (la "base") al número pasado como segundo parámetro (el "exponente", el cual puede ser incluso una fracción). Por ejemplo, si ejecutamos resultado = pow (2,5); la variable "resultado" valdrá 32 (25). Ambos parámetros son de tipo "float", y el valor devuelto es de tipo "double".

- **sq()**

  Eleva al cuadrado el número pasado como parámetro.

- **sqrt():**

  Devuelve la raíz cuadrada del número pasado como parámetro (que puede ser tanto entero como decimal). El valor devuelto es de tipo "double".

Funciones trigonométricas

- **sin():**

  Devuelve el seno de un ángulo especificado como parámetro en radianes. Este parámetro es de tipo "float". Su retorno puede ser un valor entre -1 y 1 y es de tipo "double".

- **cos():**

  Devuelve el coseno de un ángulo especificado como parámetro en radianes. Este parámetro es de tipo "float". Su retorno puede ser un valor entre -1 y 1 y es de tipo "double".

- **tan():**

  Devuelve la tangente de un ángulo especificado como parámetro en radianes. Este parámetro es de tipo "float". Su retorno puede ser un valor entre -∞ y ∞ y es de tipo "double".

Instrucciones pseudialeatorias:

- **RandomSeed()**:

  Inicializa el generador de números pseudoaleatorios. Se suele ejecutar en la sección "setup()" para poder utilizar a partir de entonces números pseudoaleatorios en nues-

tro sketch. Esta instrucción tiene un parámetro de tipo “int” o “long” llamado “semilla” que indica el valor a partir del cual empezará la secuencia de números. Semillas iguales generan secuencias iguales, así que interesará en múltiples ejecuciones de randomSeed() utilizar valores diferentes de semilla para aumentar la aleatoriedad. También nos puede interesar a veces lo contrario: fijar la semilla para que la secuencia de números aleatorios se repita exactamente.

No tiene ningún valor de retorno.

- **random()**:

  Una vez inicializado el generador de números pseudoaleatorios con randomSeed(), esta instrucción retorna un número pseudoaleatorio de tipo “long” comprendido entre un valor mínimo (especificado como primer parámetro –opcional–) y un valor máximo (especificado como segundo parámetro) menos uno. Si no se especifica el primer parámetro, el valor mínimo por defecto es 0. El tipo de ambos parámetros puede ser cualquiera mientras sea entero.

Además de las instrucciones matemáticas anteriores, el lenguaje Arduino dispone de varios operadores aritméticos. Estos operadores funcionan tanto para números enteros como decimales y son los siguientes:

- **+** Operador suma
- **-** Operador resta
- * Operador multiplicación
- **/** Operador división
- **%** Operador módulo

El operador módulo sirve para obtener el resto de una división. Por ejemplo:

27%5=2. Es el único operador que no funciona con “floats”.

Un comentario final: posiblemente, a priori las funciones matemáticas del lenguaje Arduino puede parecer escasas comparadas con las de otros lenguajes de programación. Pero nada más lejos de la realidad: precisamente porque el lenguaje Arduino no deja de ser un “maquillaje” del lenguaje C/C++, en realidad tenemos a nuestra disposición la mayoría de funciones matemáticas (y de hecho, prácticamente cualquier tipo de función) que ofrece el lenguaje C/C++. Concretamente, podemos utilizar todas las funciones listadas en la referencia online de

"avr-libc" *(http://www.nongnu.org/avr-libc/user-manual).* Así pues, si necesitamos calcular el exponencial de un número decimal "x", podemos usar exp(x);. Si queremos calcular su logaritmo natural, podemos usar log(x);. Si queremos calcular su logaritmo en base 10, podemos usar log10(x);. Si queremos calcular el módulo de una división de dos números decimales, podemos usar fmod(x,y); etc.

4.8 CREACIÓN DE INSTRUCCIONES (FUNCIONES) PROPIAS

A veces a lo largo de nuestros sketch, se puede dar el caso que repetimos varias veces el mismo conjunto de instrucciones. Para evitar esto podríamos crear una función con este conjunto de instrucciones para simplificar y clarificar el programa.

De esta forma, se puede ejecutar todo el código incluido dentro de ella simplemente escribiendo su nombre en el lugar deseado de nuestro sketch.

Al crear nuestras propias funciones, escribimos código mucho más legible y fácil de entender. Segmentar el código en diferentes funciones permite al programador crear piezas modulares de código que realizan una tarea definida.

Además, una función la podemos reutilizar en otro sketch, de manera que con el tiempo podemos tener una colección muy completa de funciones que nos permitan escribir código muy rápida y eficientemente.

Para crear una función propia, debemos "declararlas". Esto se hace en cualquier lugar fuera de "void setup()" y "void loop()" –por tanto, bien antes o después de ambas secciones– , siguiendo la sintaxis marcada por la plantilla siguiente:

tipoRetorno nombreFuncion (tipo param1, tipo param2,...) {

// Código interno de la función

}

Donde:

"tipo Retorno" es uno de los tipos ya conocidos ("byte", "int", "float", etc.) e indica el tipo de valor que la función devolverá al sketch principal una vez ejecutada. Este valor devuelto se podrá guardar en una variable para ser usada en el sketch principal, o simplemente puede ser ignorado. Si no se desea devolver ningún dato (es decir: que la función realice su tarea y punto), se puede utilizar como tipo de retorno uno especial llamado "void" o bien no especificar ninguno. Para devolver el dato, se ha de utilizar la instrucción return valor;, la cual tiene como efecto "colateral" el fin de la ejecución de la función. Esto hace que normalmente la instruc-

ción “return” sea la última en escribirse dentro del código de la función. Si la función no retorna nada (es decir, si el tipo de retorno es “void”), no es necesario escribirla.

”tipo param1, tipo param2,...” son las declaraciones de los parámetros de la función, que no son más que variables internas cuya existencia solo perdura mientras el código de esta se esté ejecutando. El número de parámetros puede ser cualquiera: ninguno, uno, dos, etc. El valor inicial de estos parámetros se asigna explícitamente en la “llamada” a la función (esto es, cuando esta es invocada dentro del sketch principal), pero este valor puede variar dentro de su código interno. En todo caso, al finalizar la ejecución de la función, todos sus parámetros son destruidos de la memoria del microcontrolador.

4.9 BLOQUES CONDICIONALES (estructuras de control)

Como ya se ha mencionado, el código de Arduino al igual que C, es un ejemplo de **programación estructurada**. En este tipo de programación, es necesario contar con ciertas estructuras que permitan controlar el flujo del programa, es decir, tomar decisiones y repetir acciones.

4.9.1 “if” y “if/else”

En la gran mayoría de los programas será necesario tomar decisiones sobre qué acciones realizar. Esas decisiones pueden depender de los datos que introduzca el usuario, de si se ha producido algún error o de cualquier otra cosa.

La estructura condicional **if ... else** es la que nos permite tomar ese tipo de decisiones. Traducida literalmente del inglés, se la podría llamar la estructura "si...si no", es decir, "si se cumple la condición, haz esto, y si no, haz esto otro".

También podemos utilizar solo la función **if**, si se cumple la condición haz esto... y sino se cumple, continua con la siguiente línea del programa después del bloque if.

Un ejemplo sencillo sería el siguiente (no se trata de un programa completo, sino tan sólo una porción de código):

```
if (edad < 18)
{
       Serial.println("No puedes acceder");
}
else
{
       Serail.println("Bienvenido");

}
```

Como se ve en el ejemplo, la estructura de un condicional es bastante simple:

```
if (condición)
 {
     sentencias_si_verdadero;
 }
else
 {
       sentencias_si_falso;
 }
```

Es posible anidar bloques “if” uno dentro de otro sin ningún límite (es decir, se pueden poner más bloques “if” dentro de otro bloque “if” o “else”, si así lo necesitamos.

Ahora que ya sabemos las diferentes sintaxis del bloque “if”, veamos qué tipo de condiciones podemos definir entre los paréntesis del “if”. Lo primero que debemos saber es que para escribir correctamente en nuestro sketch estas condiciones necesitaremos utilizar alguno de los llamados operadores de comparación u operadores lógicos, que son los siguientes.

4.9.2 Operadores de comparación

== igual que

!= distinto de

> estrictamente mayor que

>= mayor o igual que

< estrictamente menor que

<= menor o igual que

4.9.3 Operadores lógicos

Los operadores && ("y"), || ("o") y ! ("no") son operadores lógicos. Permiten operar con expresiones lógicas para generar expresiones más complejas.

4.9.4 La estructura condicional abierta y cerrada switch … case

La estructura condicional *switch ... case* se utiliza cuando queremos evitarnos las llamadas escaleras de decisiones. La estructura if nos puede proporcionar, únicamente, dos resultados, uno para verdadero y otro para falso. Una estructura *switch ... case*, por su parte, nos permite elegir entre muchas opciones.

Su sintaxis es la siguiente:

```
switch (expresión) {
                case valor1:
                //Instrucciones que se ejecutarán cuando “expresión” sea igual
        a “valor1”
                break;
```

```
        case valor2:
        //Instrucciones que se ejecutarán cuando "expresión" sea igual
    a "valor2"
        break;

        /*Puede haber los "case" que se deseen,
        y al final una sección "default" (opcional)*/
        default:
        //Instrucciones que se ejecutan si no se ha ejecutado ningún
    "case" anterior
}
```

En una sección "***case***" el valor a comparar tan solo puede ser de tipo entero. (Esto es solo para Arduino)

Las sentencias break son muy importantes, ya que el comportamiento normal de un bloque switch es ejecutarlo todo desde la etiqueta case que corresponda hasta el final. Por ello, si no queremos que se nos ejecute más de un bloque, pondremos sentencias break al final de cada bloque excepto el último.

4.9.5 El bucle while

El bloque "while" ("mientras", en inglés) es un bloque que implementa un bucle; es decir, repite la ejecución de las instrucciones que están dentro de sus llaves de apertura y cierre.

```
while (/*condicion*/) {
    /* Código */
}
```

La condición debe de ser una expresión lógica, similar a la de la sentencia

`if`. Primero se evalúa la condición. Si el resultado es verdadero, se ejecuta el bloque de código. Luego se vuelve a evaluar la condición, y en caso de dar verdadero se vuelve a ejecutar el bloque. El bucle se corta cuando la condición no se cumple.

Ejemplo: imprimir los números de 0 a 99: (es solo de ejemplo, faltaría código para poder ejecutarlo)

```
int i = 0;
while (i < 100)
{
 Serial.println(i);
 i = i + 1;
}
```

Otra aplicación práctica del bucle “while” es la eliminación de los datos en el buffer de entrada del chip TTL-UART que no queramos, de manera que se quede este limpio de “basura”. Esto lo podríamos hacer simplemente así:

```
while (Serial.available >0){

Serial.read();

}
```

4.9.6 El bucle do...while

El bucle do...while es un bucle que, por lo menos, se ejecuta una vez. Do significa literalmente "hacer", y while significa "mientras"

Su forma es esta:

```
do {
      /* CODIGO */
} while (/* Condición de ejecución del bucle */)
```

4.9.7 El bloque “for”

El bucle for es un bucle muy flexible y a la vez muy potente ya que tiene varias formas interesantes de implementarlo, su forma más tradicional es la siguiente:

```
for (/* inicialización */; /* condición */; /* incremento */) {
      /* código a ejecutar */
}
```

Inicialización: en esta parte se inicia la variable que controla el bucle y es la primera sentencia que ejecuta el bucle. Sólo se ejecuta una vez ya que solo se necesita al principio del bucle.

Expresión condicional: al igual que en el bucle `while`, esta expresión determina si el bucle continuará ejecutándose o no.

Incremento: es una sentencia que ejecuta al final de cada iteración del bucle. Por lo general, se utiliza para incrementar la variable con que se inició el ciclo. Luego de ejecutar el incremento, el bucle revisa nuevamente la condición, si es verdadera tiene lugar una ejecución más del cuerpo del ciclo, si es falsa se termina el ciclo y así.

Ejemplo: imprimir los números de 0 a 99: (es solo de ejemplo, faltaría código para poder ejecutarlo). Este ejemplo hace lo mismo que el expuesto para el bucle while.

```
int i;
for (i=0; i < 100; i = i + 1) {
      printf("%d\n", i);
}
```

4.9.8 Operadores compuestos

- x++ Equivale a x=x+1 (Al operador "++" se le llama operador "incremento")
- x-- Equivale a x=x-1 (Al operador "--" se le llama operador "decremento")
- x+=3 Equivale a x=x+3
- x-=3 Equivale a x=x-3
- x*=3 Equivale a x=x*3
- x/=3 Equivale a x=x/3

4.9.9 Las instrucciones "break" y "continue"

La instrucción "**break**" y la instrucción "**continue**" están muy relacionadas con los bucles (ya sean del tipo "while" o "for"). Observar que ninguna de ellas incorpora paréntesis, pero como cualquier otra instrucción, en nuestros sketches deben ser finalizadas con un punto y coma.

La instrucción "**break**" debe estar escrita dentro de las llaves que delimitan las sentencias internas de un bucle, y sirve para finalizar este inmediatamente. Es decir, esta instrucción forzará al programa a seguir su ejecución a continuación de la llave de cierre del bucle. En caso de haber varios bucles anidados (unos dentro de otros), la sentencia "break" saldrá únicamente del bucle más interior de ellos.

La instrucción "**continue**" también debe estar escrita dentro de las llaves que delimitan las sentencias internas de un bucle y sirve para finalizar la iteración actual y comenzar inmediatamente con la siguiente. Es decir, esta instrucción forzará al programa a "volver para arriba" y comenzar la evaluación de la siguiente iteración aun cuando todavía queden instrucciones pendientes de ejecutar en la iteración actual. En caso de haber varios bucles anidados (unos dentro de otros) la sentencia "**continue**" tendrá efecto únicamente en el bucle más interior de ellos.

5 Electrónica básica [23]

5.1 Introducción

Que es la corriente eléctrica [19]

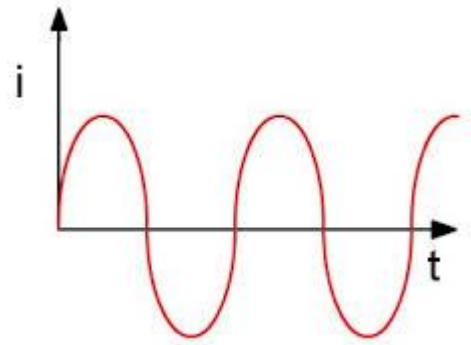
La **corriente eléctrica** o **intensidad eléctrica** es el flujo de carga eléctrica por unidad de tiempo que recorre un material. Se debe al movimiento de las cargas (normalmente electrones) en el interior del material. En el Sistema Internacional de Unidades, se expresa en C/s (culombios sobre segundo), unidad que se denomina amperio.

5.2 Corriente continua

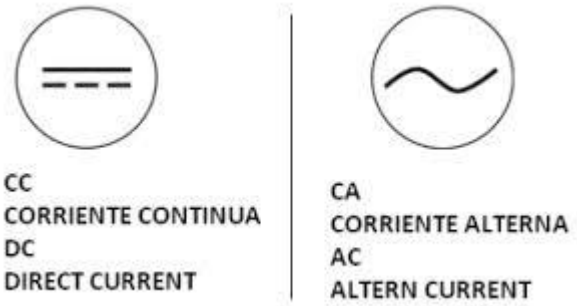
Se denomina corriente continua o corriente directa (CC en español, en inglés DC, de *Direct Current*) al flujo de cargas eléctricas que no cambia de sentido con el tiempo. La corriente eléctrica a través de un material se establece entre dos puntos de distinto potencial. Cuando hay corriente continua, los terminales de mayor y menor potencial no se intercambian entre sí. Es errónea la identificación de la corriente continua con la corriente constante (ninguna lo es, ni siquiera la suministrada por una batería). Es continua toda corriente cuyo sentido de circulación es siempre el mismo, independientemente de su valor absoluto.

5.3 Corriente alterna

Se denomina corriente alterna (simbolizada CA en español y AC en inglés, de *Alternating Current*) a la corriente eléctrica en la que la magnitud y dirección varían cíclicamente. La forma de onda de la corriente alterna más comúnmente utilizada es la de una onda sinusoidal. En el uso coloquial, "corriente alterna" se refiere a la forma en la cual la electricidad llega a los hogares y a las empresas.



Simbología:



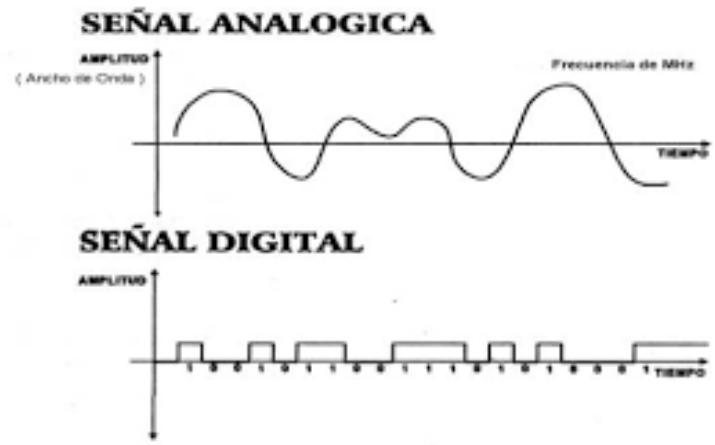
5.4 Señales analógicas y señales digitales

Señal analógica es aquella que tiene infinitos valores posibles dentro de un rango determinado (lo que se suele llamar “tener valores continuos”).

Una onda sinusoidal es una señal analógica de una sola frecuencia. Los voltajes de la voz y del video son señales analógicas que varían de acuerdo con el sonido o variaciones de la luz que corresponden a la información que se está transmitiendo.

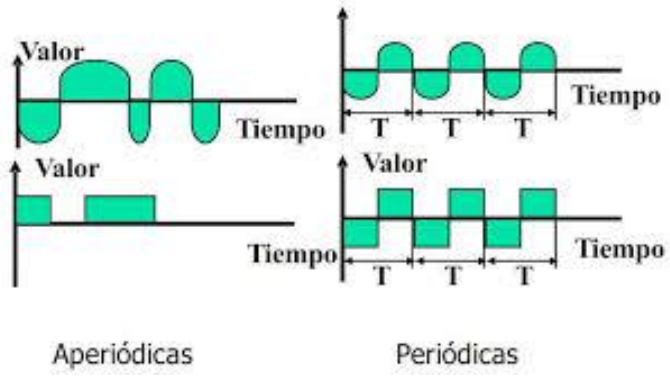
Señal digital es aquella que solo tiene un número finito de valores posibles (lo que se suele llamar “tener valores discretos”).

Un caso particular de señal digital es la señal binaria, donde el número de valores posibles solo es 2. Conocer este tipo de señales es importante porque en la electrónica es muy habitual trabajar con voltajes (o intensidades) con tan solo dos valores. En estos casos, uno de los valores del voltaje binario suele ser 0 (o un valor aproximado) para indicar precisamente la ausencia de voltaje, y el otro valor puede ser cualquiera, pero lo suficientemente distinguible del 0 como para indicar sin ambigüedades la presencia de señal. Lo habitual sobretodo trabajando con microcontroladores (Arduinos), suele ser el valor de alimentación de este, en el caso de Arduino 5V.



5.5 Señales periódicas y aperiódicas

Otra clasificación que podemos hacer con las señales eléctricas es dividirlas entre señales periódicas y aperiódicas. Llamamos señal periódica a aquella que se repite tras un cierto período de tiempo (T) y señal aperiódica a aquella que no se repite.



5.6 Ley de ohm [20]

La **ley de Ohm**, postulada por el físico y matemático alemán Georg Simon Ohm, es una ley de la electricidad. Establece que la intensidad de la corriente *I* que circula por un conductor es proporcional a la diferencia de potencial *V* que aparece entre los extremos del citado conductor. Ohm completó la ley introduciendo la noción de resistencia eléctrica *R*; esta es el coeficiente de proporcionalidad que aparece en la relación entre *I* y *V*:

$$I = V/R$$

En la fórmula, ***I*** corresponde a la intensidad de la corriente, ***V*** a la diferencia de potencial y ***R*** a la resistencia. Las unidades que corresponden a estas tres magnitudes en el sistema internacional de unidades son, respectivamente, amperios (A), voltios (V) y ohmios (Ω).

5.7 Consecuencias energéticas de la ley de Ohm: disipación y el efecto Joule

Llamamos efecto Joule al fenómeno irreversible por el cual si en un conductor circula corriente eléctrica, parte de la energía cinética de los electrones se transforma en calor debido a los choques que sufren con los átomos del material conductor por el que circulan, elevando la temperatura del mismo. Llega un momento en el que la temperatura del conductor alcanza el equilibrio térmico con el exterior, comenzando entonces a disipar energía en forma de calor.15 El nombre es en honor a su descubridor, el físico británico James Prescott Joule.

5.8 La potencia

La **potencia eléctrica** es la relación de paso de energía de un flujo por unidad de tiempo; es decir, la cantidad de energía entregada o absorbida por un elemento en un tiempo determinado. La unidad en **el Sistema Internacional de Unidades** es el **vatio** (*watt*).

P = V*I; P = R*I2

5.9 Componentes electricos [21]

5.9.1 Clasificación

Según su estructura física

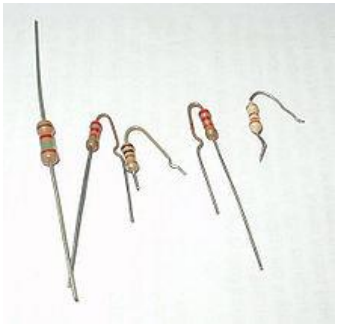
- **Discretos**: son aquellos que están encapsulados uno a uno, como es el caso de los resistores, condensadores, diodos, transistores, bobínas, etc.
- **Integrados**: forman conjuntos más complejos, como por ejemplo un amplificador operacional o una puerta lógica, que pueden contener desde unos pocos componentes discretos hasta millones de ellos. Son los denominados circuitos integrados.

Según su funcionamiento.

- **Activos**: proporcionan excitación eléctrica, ganancia o control (transistore, diodo, microcontrolador, etc).

- **Pasivos**: son los encargados de la conexión entre los diferentes componentes activos, asegurando la transmisión de las señales eléctricas o modificando su nivel (resistencias, condensadores y bobinas).

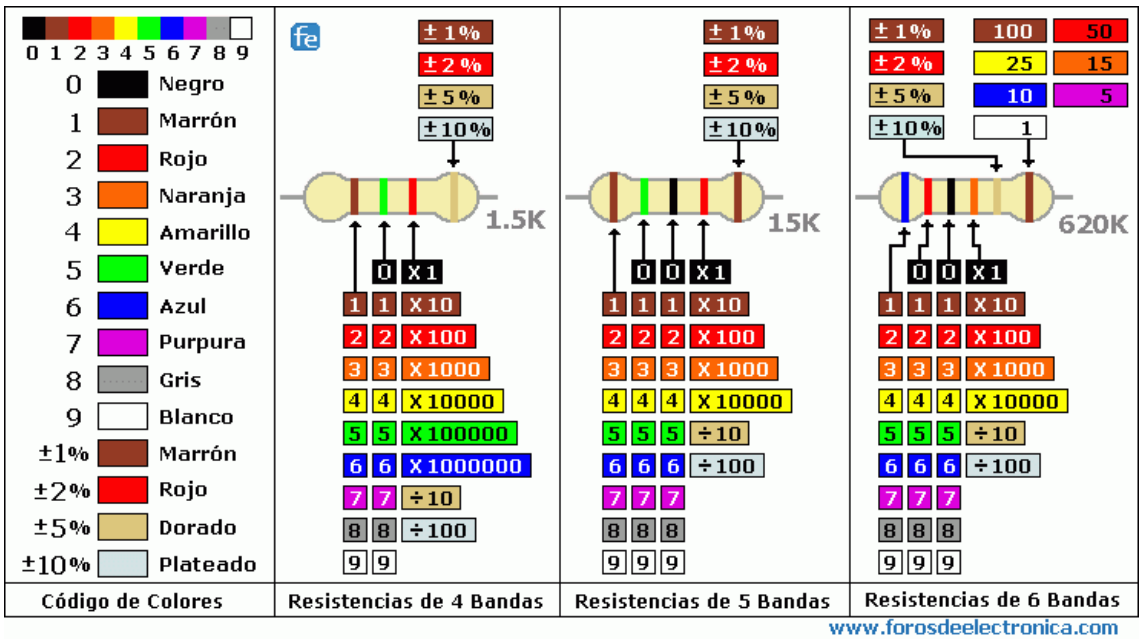
5.9.2 Resistencias [22]



Ya hemos visto que las resistencias son componentes discretos y pasivos.

Los resistores se utilizan en los circuitos para limitar el valor de la corriente o para fijar el valor de la tensión. A diferencia de otros componentes electrónicos, los resistores no tienen polaridad definida.

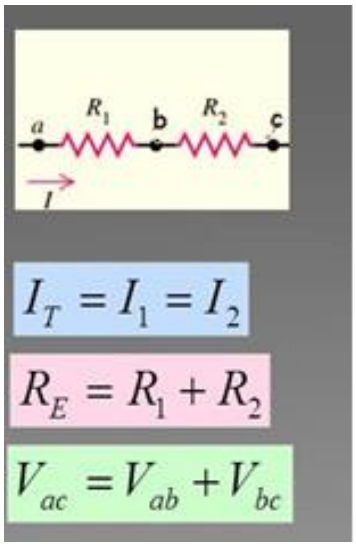
Para distinguir sus valores, en el caso de las resistencias de tipo axial (ver figura anterior) es mediante un código de colores, representado en la siguiente tabla:



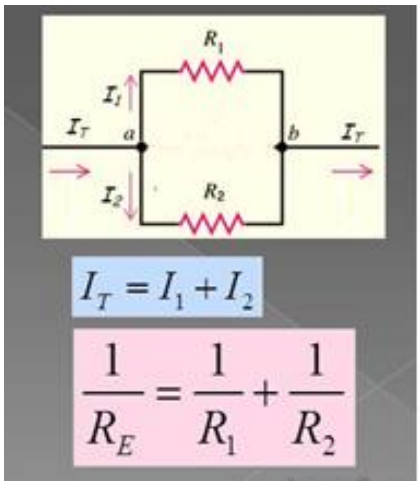
5.9.2.1 *Resistencias en serie y en paralelo*

La asociación de resistencias se puede sustituir por una resistencia equivalente. Representada por *RE*

- Resistencias en **serie**:

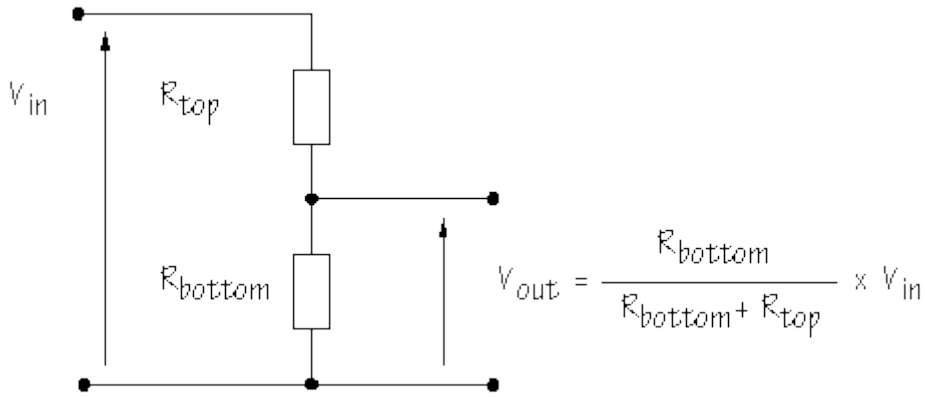


- Resistencias en **paralelo**:



5.9.2.2 Divisor de tensión.

Un divisor de tensión es una configuración de circuito eléctrico que reparte la tensión de una fuente entre una o más impedancias conectadas en serie.

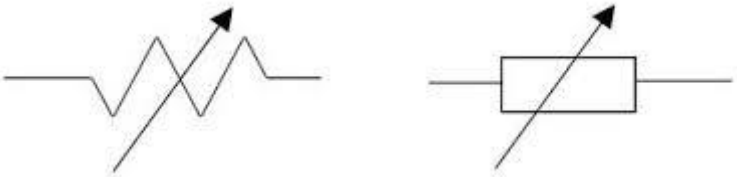


R_{top} y R_{bottom} pueden ser cualquier combinación de resistencias en serie o paralelo.

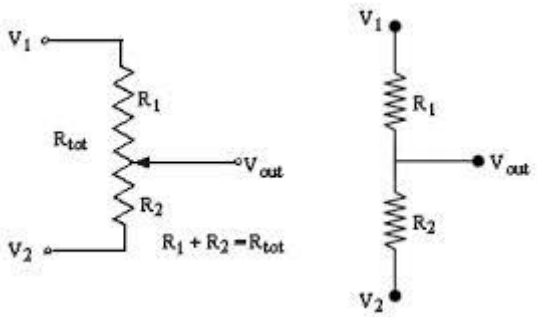
5.9.2.3 Potenciómetro

Un **potenciómetro** es un resistor cuyo valor de resistencia es variable. De esta manera, indirectamente, se puede controlar la intensidad de corriente que fluye por un circuito si se conecta en paralelo, o la diferencia de potencial al conectarlo en serie.

Simbología:



Si analizamos con detalle el funcionamiento del potenciómetro, vemos que es un divisor de tensión.



5.9.3 Condensador

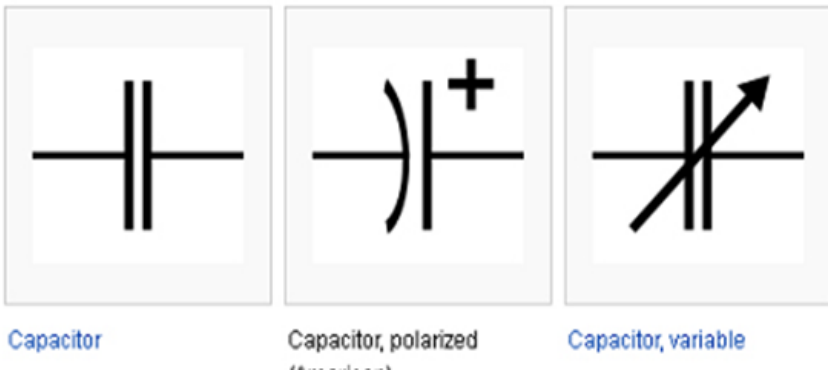
Un condensador (en inglés, capacitor), es un dispositivo pasivo, utilizado en electricidad y electrónica, capaz de almacenar energía sustentando un campo eléctrico.

Los hay de diversos tipos: cerámicos, de poliéster, electrolíticos, de papel, de mica, de tántalo, variables y ajustables.

Las utilidades más comunes, son:

- Para fuentes de alimentación, estabilización de tensiones.
- Filtros,
- adaptación de impedancias.
- Resonadores
- etc

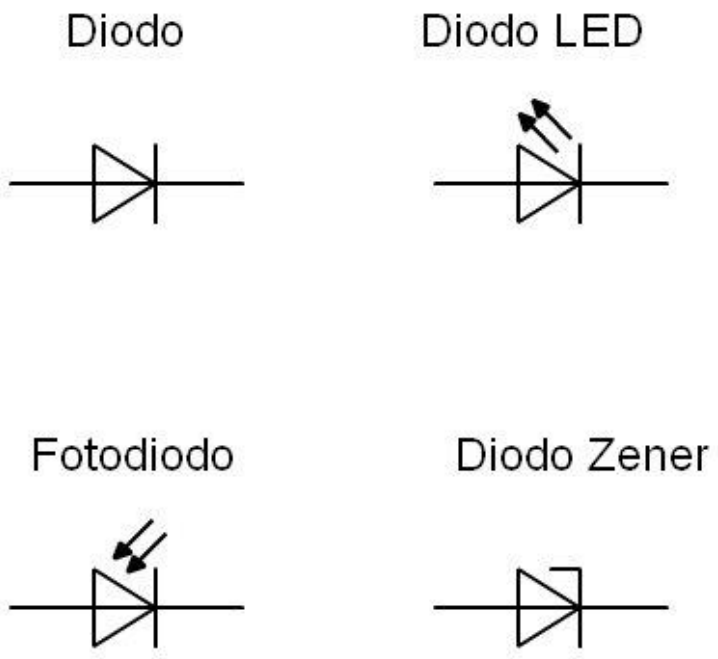
Lo más importante es saber que los electrolíticos (de los más utilizados), tienen polaridad y si no se respeta, el condensador explota.



5.9.4 Diodo

El diodo es un componente electrónico de 2 terminales, tal como un resistor.

Hay varios tipos:

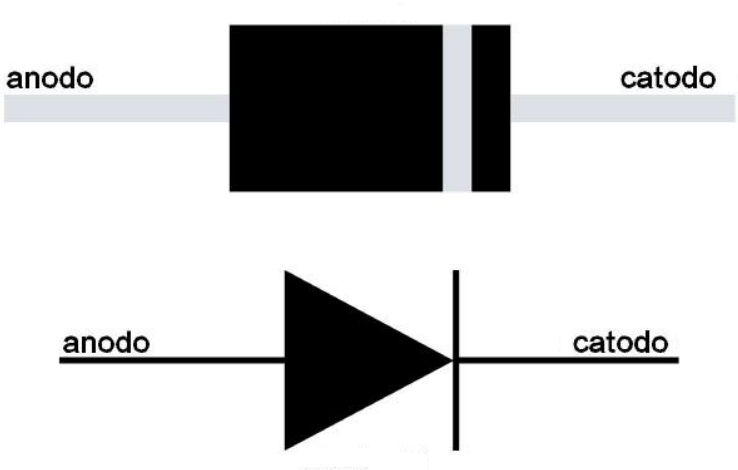


Los más usados, sobretodo en proyectos de Arduino será, el diodo rectificador (el diodo normal) y el diodo led.

5.9.4.1 Diodo rectificador:

Un diodo es un dispositivo diseñado para que la corriente fluya en un solo sentido, es decir, solamente permite que la corriente vaya en una sola dirección.

El símbolo representativo del diodo en esquemas electrónicos es el siguiente:



La corriente fluye desde el terminal positivo (el ánodo) hasta el terminal negativo (cátodo).

En los diodos físicos se identifica el cátodo por una franja que se coloca en uno de los extremos del diodo.

5.9.4.2 Utilidades

- **Impedir que una corriente fluya en un sentido no deseado** (como protección)

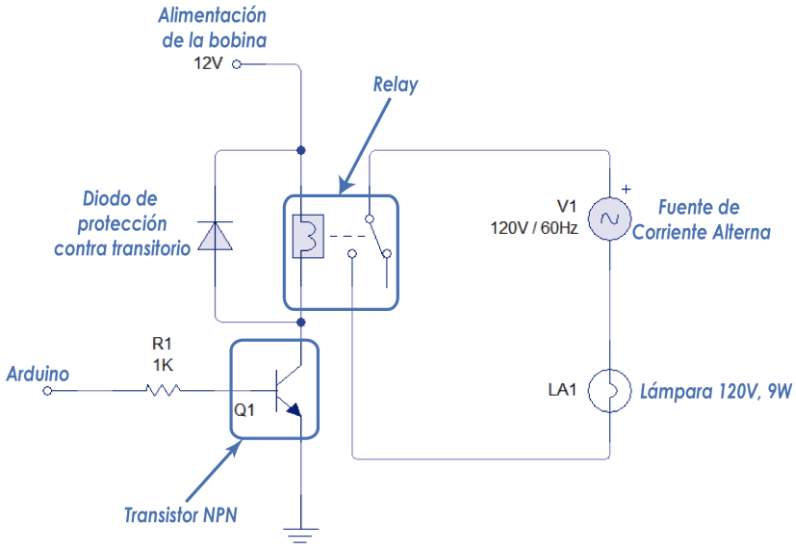
  Si por ejemplo utilizamos Arduino para conmutar un circuito con un voltaje superior al soportado por el microcontrolador y queremos impedir que cualquier corriente fluya hacia Arduino, es decir, si queremos lograr que Arduino entregue una tensión y bajo ninguna circunstancia se exponga a una corriente de entrada, entonces se utiliza un diodo entre Arduino y el dispositivo a usar.

  Al impedir que haya un flujo de corriente hacia Arduino se protege al microcontrolador de que una posible sobre corriente termine destruyendo el microcontrolador.

- **Suprimir corrientes transitorias**

  En ocasiones cuando los circuitos tienen bobinas (inductores) o condensadores se producen corrientes transitorias. Estas corrientes afectan ciertas partes del circuito, por lo que se trata de suprimirlas para que no causen problemas.

  Un uso muy común de esto, es para evitar corrientes transitorias al activar un relé. [24]



- **Provocar caídas de voltaje**

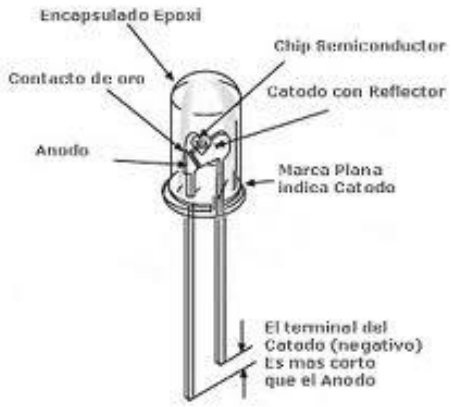
  Cuando se conecta un diodo en un circuito inmediatamente se produce una caída de voltaje de 0.7 voltios entre el ánodo y el cátodo. Esta caída de 0.7 voltios podría ser utilizada para diferentes propósitos como conversión análogo/digital por ejemplo.

- **Rectificar corriente alterna**

Podría decirse, que es su uso más habitual, ya que es la base de toda fuente de tension de continua, a partir de una corriente alterna.

5.9.4.3 El diodo led

Básicamente es igual, con la diferencia de que cuando circula corriente a través de él, este emite luz.



5.9.5 Transistores

El **transistor** es un dispositivo electrónico semiconductor utilizado para entregar una señal de salida en respuesta a una señal de entrada. Cumple funciones de **amplificador**, oscilador, **conmutador** o rectificador. El término «transistor» es la contracción en inglés de *transfer resistor* («resistencia de transferencia»). Actualmente se encuentran prácticamente en todos los aparatos electrónicos de uso diario: radios, televisores, reproductores de audio y video, relojes de cuarzo, computadoras, lámparas fluorescentes, tomógrafos, teléfonos celulares, entre otros.

Los transistores pueden tener tres estados posibles de trabajo dentro de un circuito (tanto los bipolares como los de efecto campo. (ver tipos)

- En activa: deja pasar más o menos corriente.
- En corte: no deja pasar la corriente.
- En saturación: deja pasar toda la corriente.

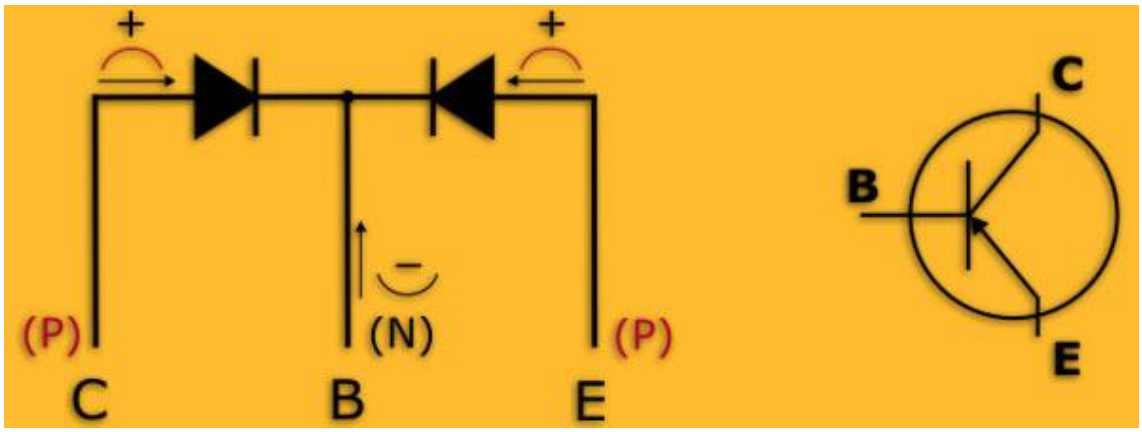
5.9.6 Tipos de transistores

5.9.6.1 Transistor de unión bipolar [25] [26]

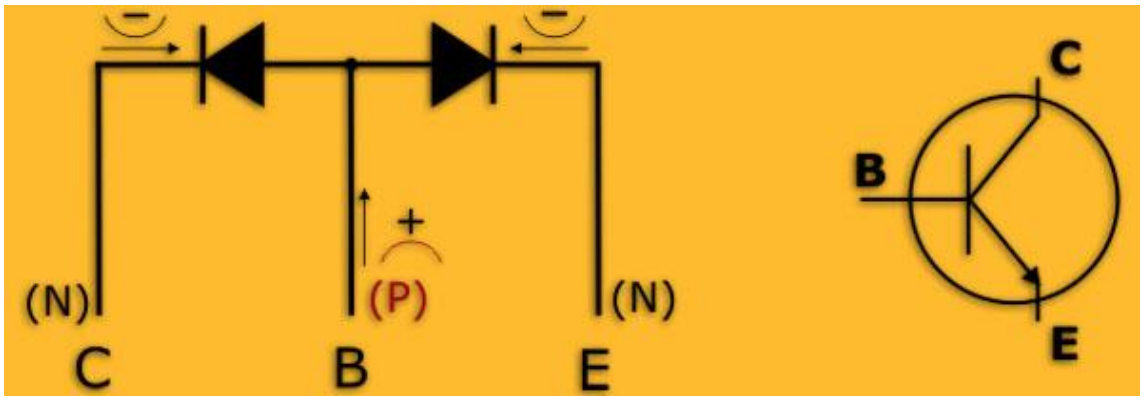
El transistor de unión bipolar (o BJT, por sus siglas del inglés *Bipolar Junction Transistor*).

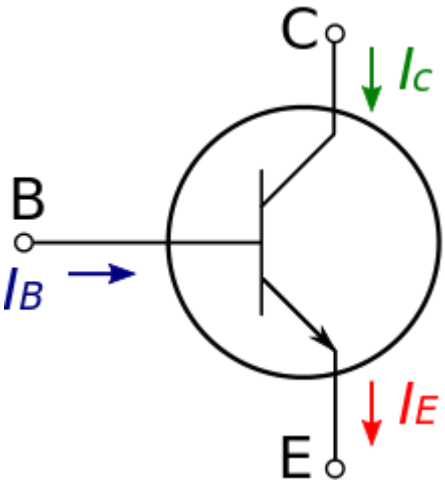
Puede ser NPN o PNP

PNP



NPN





Las fórmulas que relacionan dichas corrientes son:

IE = IC + IB;

Como IB es muy bajo comparado con IC, se suele decir que : IE = IC;

Siendo IC:

$I_C = \beta I_{B;}$

Donde β (hfe) es el factor de amplificación (ganancia) del transistor. (Dato dado por el fabricante)**.**

La ganancia es realmente lo que se amplifica la corriente en el transistor. Por ejemplo una ganancia de 100 significa que la corriente que metamos por la base se amplifica, en el colector, 100 veces, es decir será 100 veces mayor la de colector que la de la base. Como la de colector es muy parecida a la del emisor, podemos aproximar diciendo que la corriente del emisor también es 100 veces mayor que la de la base.

En un transistor que tenga una ganancia de 10 si metemos 1 amperio por la base, por el colector obtendremos 10 amperios. Como ves es el transistor también es un amplificador. Pero OJO imagina que el transistor que tienes solo permite como máximo 5 amperios de salido, ¿qué pasaría si metemos 1 amperio en la base? ¡¡¡Se quemaría!! porque no soportaría esa corriente en el colector.

También es muy importante que sepas que la corriente del colector depende del receptor que tengamos conectado a la salida, entre el colector y el emisor. La corriente del colector será la que "chupe" ese receptor, nunca mayor. Si en el caso anterior el receptor fuera un lámpara que solo consumiera 3 amperios no pasaría nada, ya que entre el emisor y el colector solo circularían los 3 amperios que demanda la lámpara.

Otra cosa a tener en cuenta, sobretodo trabajando con Arduino es que si queremos mover un motor utilizando un JBT y este consume mucha corriente, debemos de saber que la corriente de la base IB, no debe de superar 40mA, (corriente máxima suministrada por las salidas digitales del Arduino.

5.9.6.2 Transistor de efecto de campo [27]

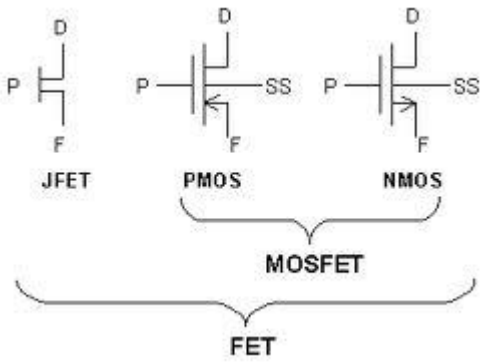
Dentro de los denominados transistores de efecto campo (FET), podemos tener: JFET, IGFET Y MOSFET (estos últimos los más utilizados)

Al igual que los JBT, los de tipo FET, pueden ser NPN ó PNP

El MOSFET

El **transistor de efecto de campo metal-óxido-semiconductor** o **MOSFET** (en inglés *Metal-oxide-semiconductor Field-effect transistor*) es un transistor utilizado para amplificar o conmutar señales electrónicas. Es el transistor más utilizado en la industria microelectrónica, ya sea en circuitos analógicos o digitales, aunque el transistor de unión bipolar fue mucho más popular en otro tiempo. Prácticamente la totalidad de los microprocesadores comerciales están basados en transistores MOSFET.

El MOSFET es un dispositivo de cuatro terminales llamados surtidor (S), drenador (D), compuerta (G) y sustrato (B). Sin embargo, el sustrato generalmente está conectado internamente al terminal del surtidor, y por este motivo se pueden encontrar dispositivos MOSFET de tres terminales.



5.10 Los drivers [28][29]

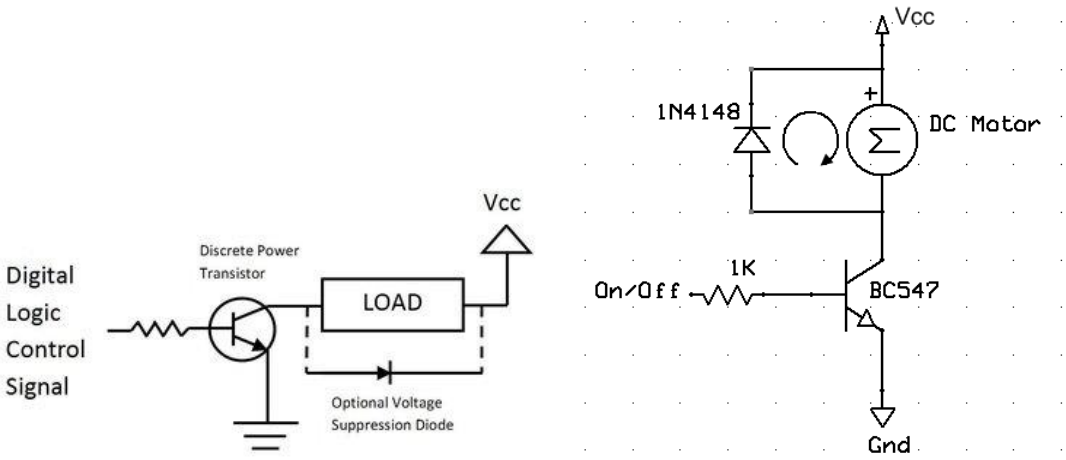
En electrónica, un driver es un circuito eléctrico o componente eléctrico utilizado para controlar otro circuito o componente eléctrico.

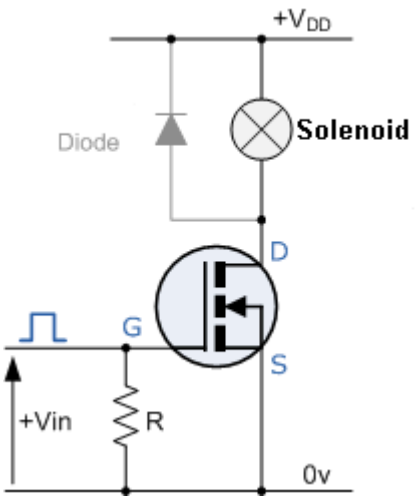
Muchas veces nos encontraremos con la situación de que tenemos que controlar un dispositivo que consume mucha más corriente de la que nos puede proporcionar las salidas digitales de Arduino. Para esos casos necesitaremos intercalar un circuito o componente electrónico, que nos permita controlar altas corrientes para el control dicho dispositivo. Estos circuitos o componentes eléctricos son los denominados drivers.

La unidad más básica que podemos utilizar para construir dicho driver sería un transistor, trabajando como interruptor.

Este puede ser o un BJT o un MOSFET. La mayor diferencia entre ellos, es que el MOSFET no consume corriente y nos permite controlar mayores potencias, pero es más caro.

El montaje de estos como driver sería:





Hemos comentado anteriormente que estos drivers son los más básicos que se pueden tener, pero existen otros como opto acopladores **[30]** o circuitos integrados los cuales internamente están compuestos por transistores, para permitirnos controlar corrientes más altas con señales muy débiles.

Bibliografía

- [1] http://arduino.cc/en/Guide/Introduction
- [2] http://es.wikipedia.org/wiki/Microcontrolador
- [3] http://es.wikipedia.org/wiki/Creative_Commons
- [4] http://arduino.cc/en/Main/Products
- [5] http://burutek.org/es/arduino/
- [6] http://arduino.cc/en/Tutorial/ArduinoToBreadboard
- [7] http://es.wikipedia.org/wiki/Universal_Asynchronous_Receiver-Transmitter
- [8] http://arduino.cc/en/Hacking/DFUProgramming8U2
- [9] http://www.ardumania.es/reflashear-el-8u2/
- [10]http://pdf1.alldatasheet.es/datasheet-pdf/view/174874/ONSEMI/NCP1117ST50T3G.html
- [11] http://arduino.cc/en/Reference/AnalogReference
- [12] http://es.wikipedia.org/wiki/I%C2%B2C
- [13] http://arduino.cc/es/Guide/HomePage
- [14] http://www.marlonj.com/blog/2011/06/ques-es-wiring/
- [15] http://playground.arduino.cc/ArduinoNotebookTraduccion/Structure
- [16] http://arduino.cc/en/Reference
- [17] http://es.wikipedia.org/wiki/ASCII
- [18] http://arduino.cc/en/Reference/Serial
- [19] http://es.wikipedia.org/wiki/Corriente_el%C3%A9ctrica
- [20] http://es.wikipedia.org/wiki/Ley_de_Ohm
- [21] http://es.wikipedia.org/wiki/Componente_electr%C3%B3nico
- [22] http://www.forosdeelectronica.com/tutoriales/resistencia.htm

- [23] http://www.portaleso.com/usuarios/Toni/web_electronica_3/electronica_indice.html#condensador
- [24] http://panamahitek.com/conceptos-basicos-de-electronica-el-diodo/
- [25] http://es.wikipedia.org/wiki/Transistor_de_unión_bipola
- [26] http://www.areatecnologia.com/TUTORIALES/EL%20TRANSISTOR.htm
- [27] http://hispavila.com/3ds/atmega/mosfets.html
- [28] http://www.diarioelectronicohoy.com/blog/controladores-basicos-drivers
- [29] http://developer.mbed.org/users/4180_1/notebook/relays1/
- [30] http://es.wikipedia.org/wiki/Optoacoplador

Bibliografía recomendada.

Sobre programación:

- http://en.wikibooks.org/wiki/C_Programming
- http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Introducci%C3%B3n
- http://www.elrincondelc.com/cursoc/cursoc3.html

Sobre Arduino:

- Taller de Desarrollo de Sistemas Domóticos Basados en Arduino (José Luis Poza Luján. Sergio Sáez Barona
- ARDUINO Curso práctico de formación. Óscar Torrente Artero. Editorial Alfaomega.
- http://www.arduteka.com

Sobre Electrónica básica:

- http://panamahitek.com/category/electronica/conceptos-de-electronica/