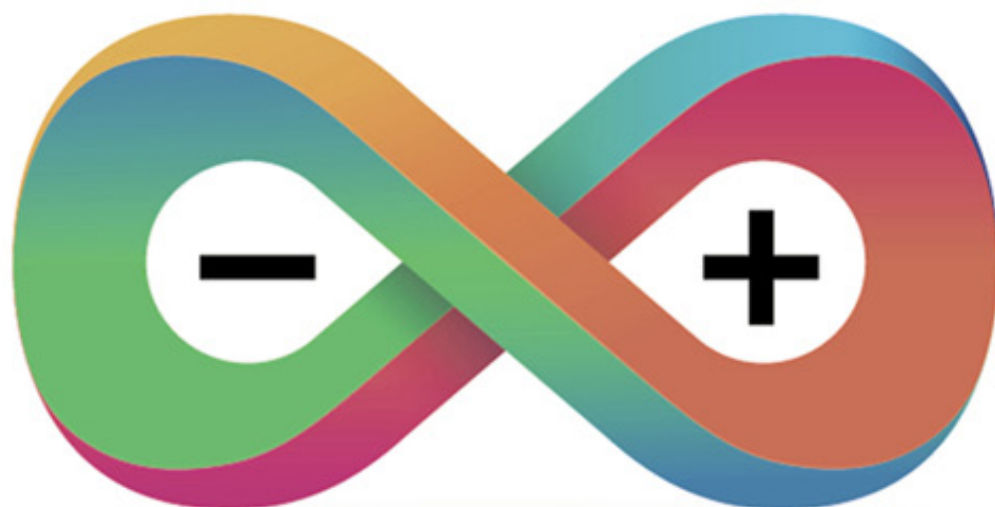


# ARDUINO

PARA JOVENES...  
Y NO TAN JÓVENES



JOAN RIBAS LEQUERICA

ANAYA  
MULTIMEDIA

# ARDUINO

PARA JÓVENES...  
Y NO TAN JÓVENES

JOAN RIBAS LEQUERICA



# Índice de contenidos

Agradecimientos  
Sobre el autor

## Capítulo 1. Cómo usar este libro

Destinatarios de este libro  
Organización del libro  
Convenios empleados  
Ejemplos del libro

## Capítulo 2. Introducción

Prototipos  
Arduino  
Arduino Uno  
Shields  
Arduino Uno  
Instalación del entorno de programación  
Entorno de programación  
Protoboard o breadboard

## Capítulo 3. Electricidad y electrónica

¿Qué es la electricidad?  
Voltaje  
Corriente o intensidad  
Resistencia  
Relación voltaje, corriente y resistencia  
Abstracción del circuito

¿Qué es la electrónica?  
Electrónica digital  
Binario  
Operaciones a nivel bit

## **Capítulo 4. Programación**

General  
Formato  
Las variables  
Operaciones  
Bloques de control  
if  
switch  
goto  
Bucles  
Funciones

## **Capítulo 5. Primeros programas**

Primer programa  
Un poco más de programación  
Control mediante if  
Control mediante switch  
Control mediante for  
Control mediante while  
Control mediante do...while  
Convertor de base

## **Capítulo 6. Entradas y salidas**

Entradas  
Entradas digitales  
Entradas analógicas  
Salidas  
Salidas digitales  
Salidas analógicas

## **Capítulo 7. Sensores**

Fotorresistencia  
Termistores  
Sensor de humedad  
Sensor de humedad sin librería  
Otros sensores

## **Capítulo 8. Salida de audio**

Altavoz piezoeléctrico como sensor  
Alarma  
Piano

## **Capítulo 9. Actuadores**

Motores  
Servomotores  
Relés  
Control de una lámpara

## **Capítulo 10. Internet**

Acceso a webs  
Otras conexiones  
Publicación Twitter  
Arduino como servidor

*A mi hija Mar.  
Sigue curioseando, probando cosas,  
investigando, jugando... y sobre todo  
que no se te olvide sonreír y ser feliz.*

## Agradecimientos

A mis hermanas por estar siempre ahí.

A mi mujer Laia porque dijo sí.

A los que me brindaron oportunidades.

A los que saben que 2 es 10.

A mi familia natural y política; en esto, sí he tenido suerte.

A los que enseñan y a los que quieren aprender.

A los que nos quieren juntos y no divididos.

## Sobre el autor

**Joan Ribas Lequerica** cursó Ingeniería Electrónica e Ingeniería de Telecomunicaciones en la Universidad de Valladolid, así como Ingeniería de Organización Industrial en Barcelona. Ha sido colaborador en proyectos GNU como Gentoo o Enlightenment. En los últimos años ha trabajado en múltiples proyectos de desarrollo de aplicaciones móviles para el sector empresarial.

Es autor de numerosas publicaciones en revistas tecnológicas y libros. Está considerado como uno de los autores técnicos de referencia en España con títulos entre los que se encuentran *Programación en Delphi*, la Guía Práctica *Web Services* y los reconocidos Manuales Imprescindibles de *Arduino Práctico* y *Desarrollo de aplicaciones para Android* en sus diferentes versiones.

# 1

## Cómo usar este libro



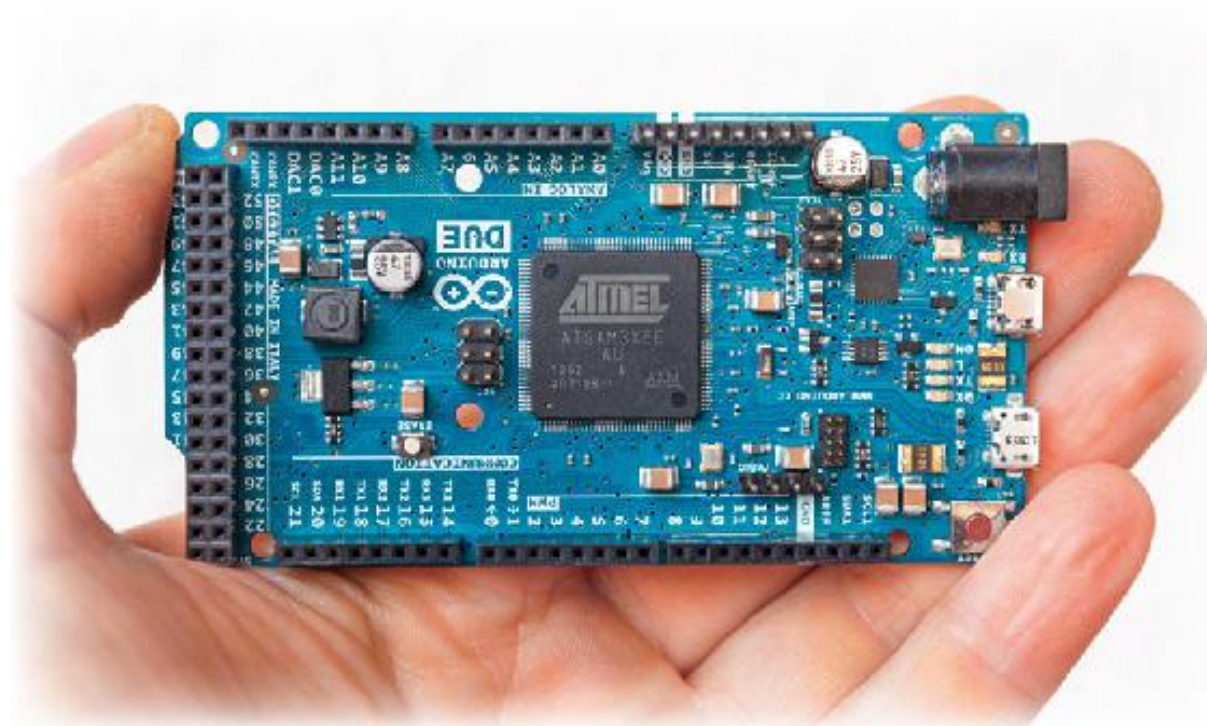


## Destinatarios de este libro

Este libro está dirigido a todos aquellos jóvenes y no tan jóvenes que se quieran iniciar en el mundo de la electrónica y más concretamente en el mundo la creación de proyectos con Arduino.

En este libro se tratarán también los aspectos básicos de la electricidad y la electrónica con tal poner las bases y acercar los circuitos y diseños electrónicos a todas las personas, incluso a los que no hayan tenido contacto previo con la electricidad o la programación.

Aunque trabajaremos con cantidades de electricidad muy pequeñas y parezca que no hay riesgo, no tenemos que olvidar que estaremos trabajando con electricidad y a la electricidad siempre hay que mirarla con respeto, por lo que recomendamos que los más pequeños estén supervisados por un adulto.



## Organización del libro

El manual que tiene entre sus manos se encuentra organizado en capítulos con ejemplos prácticos. En cada uno de ellos se explican distintos elementos electrónicos o funcionalidades de Arduino.

Se trata de un libro eminentemente práctico, es decir que los capítulos vienen acompañados de distintos ejemplos que nos serán de utilidad para afianzar conocimientos y descubrir cómo programar distintos aspectos de Arduino; es por esto que animo al lector a realizar los ejemplos paso a paso a la vez que vaya leyendo los capítulos. No obstante, todos los ejemplos se encuentran disponibles en la web de Anaya Multimedia.

Durante los capítulos, se irán introduciendo distintos conceptos relativos a la propia placa Arduino, a los componentes que se pueden montar a su alrededor para darle funcionalidad y a la programación necesaria para hacer del conjunto un elemento funcional.

Con los ejercicios se irán presentando distintos componentes electrónicos de los cuales se mostrará tanto su funcionamiento como su utilidad, sin entrar mucho en profundidad pero sí para tener una idea de cómo poder usarlos. En ocasiones las explicaciones se acompañarán de esquemas de circuitos; no se pretende que el lector se convierta en un experto analizador de circuitos, sino facilitar las explicaciones y familiarizarse con los símbolos usados en los esquemas.

No es necesario conocimientos previos de electrónica o programación ya que se explicará paso a paso.

En cuanto a los materiales necesarios, lo esencial es un ordenador (con Linux, OSX o Windows) y una tarjeta Arduino. Además para los ejemplos se necesitarán diferentes componentes electrónicos, que pueden encontrarse fácilmente en tiendas de electricidad y electrónica.

## Convenios empleados

A lo largo del texto del libro, se utilizarán ciertos convenios de tipos de letra y formatos para facilitar la lectura.

Para los esquemas existen varias notaciones entre las cuales se encuentra la IEC(*International Electrotechnical Commission*, Comisión electrotécnica internacional) pero en lugar de seguir sus recomendaciones, usaremos en cada momento la notación que más sencilla resulte para entender el circuito (dentro de la sencillez de todos ellos, no hay que asustarse).

Debido a que parte de las herramientas utilizadas solamente están disponibles en inglés, en el texto se escribirán los nombres de botones, menús, etc. en inglés, y se proporcionará traducción en todo caso.

Los nombres de botones, herramientas y combinaciones de teclas aparecen en negrita para facilitar su identificación; por ejemplo, el botón **Finish**.

Los nombres de cuadros de diálogo, menús, submenús y cualquier elemento que pueda encontrar en pantalla excepto los botones aparecen en negrita y subrayado para facilitar su identificación, por ejemplo, el menú Project.

Los accesos a los menús de la aplicación aparecen separados por el signo mayor que (>) y en el orden de la selección. Por ejemplo, File>Preferences.

Elementos como funciones, palabras clave del lenguaje, comandos de programación y en general todo lo relativo a código que se encuentre incrustado en texto explicativo, aparecerá destacado con un subrayado,

Temas de especial relevancia como advertencias o trucos se encuentran marcados mediante

---

**NOTA:** Cuadro de nota para resaltar algo importante.

---

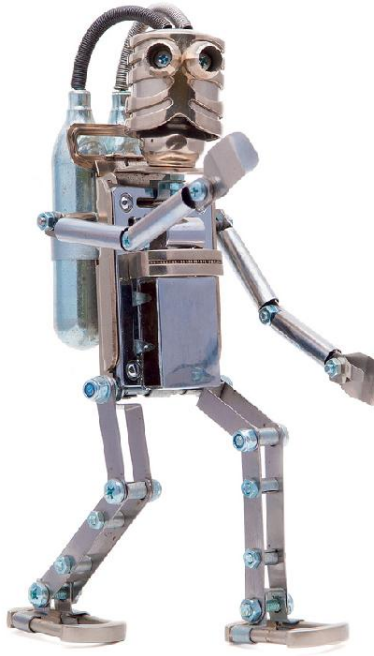
## **Ejemplos del libro**

En los ejemplos que se verán a lo largo del libro, se ha utilizado una placa Arduino UNO rev 3.0. Si el lector va a utilizar otro modelo de placa compatible, deberá tener cuidado en la asignación de los pines (los pinchos) y en la selección del dispositivo en el entorno de programación.

Todos los ejemplos que se ven en el libro se encuentran disponibles en la página web de Anaya Multimedia (<http://www.anayamultimedia.es>) en la opción Complementos de la ficha correspondiente a este libro. Puede realizar la búsqueda de la ficha a través de la búsqueda sencilla o avanzada y sus diferentes opciones.

# 2

## Introducción



### **En este capítulo aprenderá:**

- Qué es un prototipo
- Qué es una tarjeta Arduino
- Anatomía de la Arduino Uno
- Usar un *protoboard*
- Instalar el entorno de programación

# Prototipos

Los productos electrónicos no son producidos directamente como los conocemos y nos venden en las tiendas. Da igual que sea complejo como un teléfono móvil o simple como una batidora; todo producto necesita una fase inicial donde se reproduce de manera genérica lo que se quiere conseguir, pero sin tener en cuenta aspectos como la forma final. Lo que se hace es realizar pruebas de cómo se debería producir y así poder detectar fallos de diseño antes de ponerse a fabricar grandes cantidades del producto y que luego no funcionen de la manera esperada. Estas primeras pruebas es lo que se llaman prototipos.

Para hacernos una idea a grandes rasgos del proceso de fabricación, imaginemos que queremos producir un semáforo (más adelante haremos un prototipo de él); las fases por las que pasaríamos serían:

- **La idea:** Surge normalmente para dar solución a algún problema existente o para mejorar alguna solución previamente dada. Alguien piensa que los cruces de las carreteras son un peligro y que de alguna manera se debe controlar qué carriles pueden circular, de modo que cuando unos carriles puedan pasar, los carriles con los que puedan chocar no tengan permiso de circular y viceversa, dando paso alternativo a cada uno de ellos.
- **Abstracción de la idea:** Hay que encontrar la esencia de la idea, de modo que más adelante pueda ser modelada mediante electrónica. En nuestro caso, interesa poner una señal que de paso o no a los coches, simplemente esto, ya se verá qué tipo de señal debe mostrar o cuando, pero lo que se quiere es algo que muestre a los coches si se puede pasar o no.
- **Modelado:** Consiste en dar una solución que satisfaga el punto anterior. Puede haber varias soluciones y hay que seleccionar la más

óptima, que dependerá de muchos factores, por ejemplo en el caso del semáforo se puede modelar con el semáforo por todos conocido de luces, o mecánicos como existían anteriormente o poner unas barreras... supongamos que decidimos modelarlo con una serie de pinchos que estén activos cuando no se pueda pasar para que revienten las ruedas y no estén activos cuando sí se pueda pasar... efectivo es, pero no es funcional, no es la solución más óptima.

- **Creación del prototipo:** Se realiza un diseño del prototipo para implementar el modelo que se ha elegido para dar solución al problema. En nuestro caso, previos cálculos que ya veremos, se debe crear una caja con luces y un controlador que sepa qué luces se deben encender en cada momento y hacer que en el cruce no se puedan encontrar todos los semáforos en verde a la vez. El prototipo nos servirá para comprobar que durante los pasos de abstracción de la idea y modelado no nos hemos olvidado de algún tema o queden casos por satisfacer. Por ejemplo si se ha pensado el semáforo solo con una luz roja y una verde, a lo mejor es que no se ha pensado que nada más ponerse un semáforo en rojo no se puede poner el otro en verde, porque siempre hay conductores que se lo saltan nada más ponerse en rojo, por lo que hay que dar un tiempo de seguridad antes de poner el otro semáforo en verde, o quizá mejor añadir una tercera luz naranja para avisar que se va a poner en rojo. El prototipo también sirve para comprobar que los elementos seleccionados para el montaje son correctos, por ejemplo si las bombillas del semáforo tienen la duración correcta o si los materiales no se oxidan al estar en contacto con la lluvia, lo que ahorra mucho dinero en la creación del producto final... Esta fase es muy importante para el futuro éxito del funcionamiento del producto final, por lo que se le debe prestar toda la atención que merece y no debe olvidarse nunca.
- **Creación de fotolitos:** Una vez comprobado que el circuito creado para el prototipo funciona de la manera esperada, se deben generar “los planos” del circuito para que puedan ser producidos a escala comercial de manera automática, esto se hace mediante un programa de ordenador que sea capaz de colocar los elementos necesarios del

circuito en una superficie y crear las pistas de unión entre ellos (los cables de nuestro prototipo) de modo que no se crucen. El resultado del posicionamiento y camino de las conexiones será un dibujo de las pistas que se enviará para la producción de la placa final; son los llamados fotolitos.

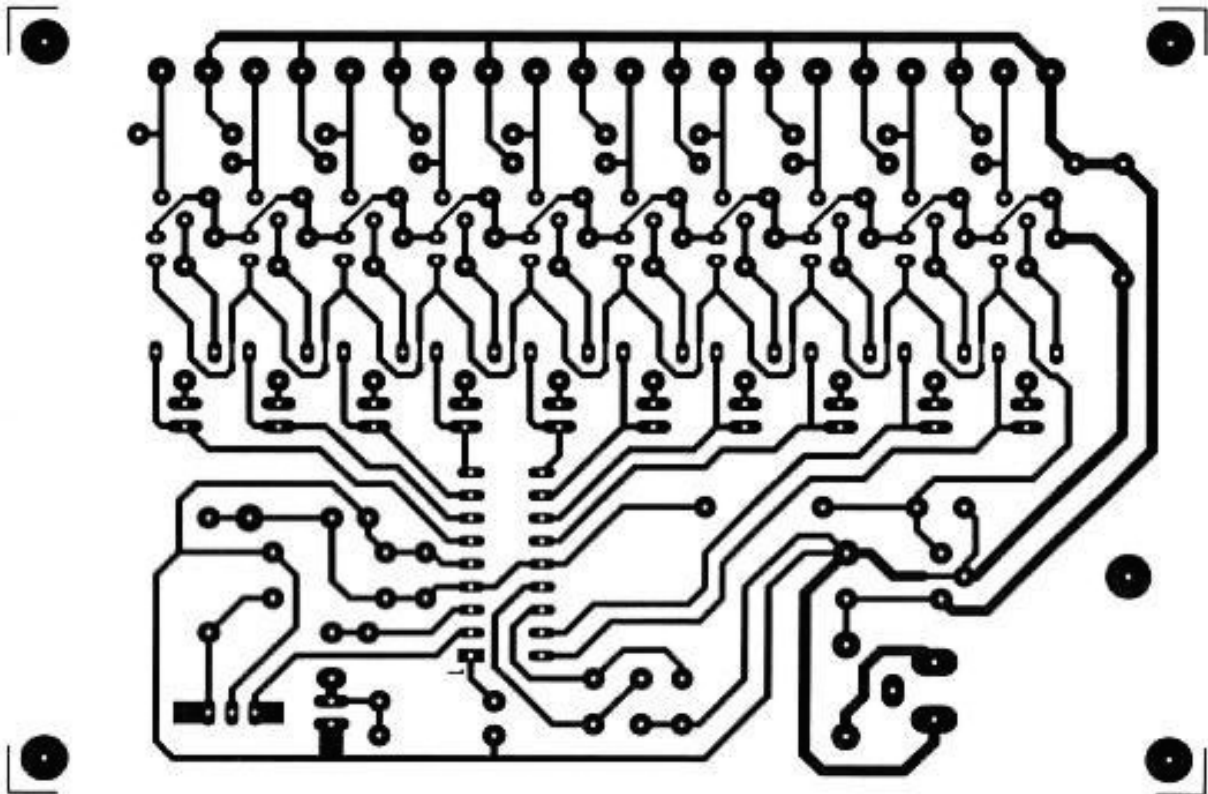


Figura 2.1. Fotolito de un circuito

- **Unión de componentes:** La placa del circuito creada a partir del fotolito, no deja de ser una superficie plana que no conduce la electricidad con unas pistas conductoras dibujadas sobre ella y que unirán los componentes electrónicos. Lo normal es que nos encontremos unos agujeros en las pistas, éstos son para colocar en ellos las patillas de los componentes electrónicos para poder soldarlos. También podemos encontrar circuitos donde no existen estos agujeros y lo que se hace es que el componente simplemente se apoya en las pistas y es fijado así sobre ellas. Para fijar los componentes a la placa se utiliza una mezcla de estaño y plomo,



aunque por ser el plomo peligroso se está intentando sustituir por otros componentes.

En este libro nos centraremos en el punto de creación del prototipo, que es donde utilizaremos Arduino, pudiendo crear objetos que más adelante podrían ser producidos a nivel industrial.

## Arduino

Podemos decir que Arduino es una familia de tarjetas electrónicas que, acompañadas de un software, nos permiten realizar prototipos electrónicos o si queremos explicarlo de una manera más sencilla, podríamos ver a la placa Arduino como un pequeño ordenador al que se le puede dar órdenes (programar) para que interactúe con el mundo real, bien ofreciendo salidas o reaccionando a una serie de entradas. Existen muchos tipos de tarjetas Arduino (y copias de otras marcas) y dependiendo del modelo podemos tener más o menos funcionalidades.

Tanto las entradas que puede entender la tarjeta como las salidas que ofrece, son de naturaleza eléctrica, dicho de otro modo, la tarjeta Arduino solo entiende electricidad, entonces, si queremos hacer un termómetro, necesitaremos leer la temperatura de alguna manera, transformar el valor en electricidad, procesarlo mediante Arduino y valernos de algún otro medio para volver a transformar el valor eléctrico en algo utilizable por el mundo real (si es un humano por ejemplo mediante una pantalla o si es una máquina como podría ser el sistema de calefacción, adecuar la salida eléctrica a lo esperado por ella).

Para poder decir a la placa lo que tiene que hacer, le tenemos que dar unas órdenes, es decir se necesita programarla, pero no entiende nuestro idioma, sino uno especial llamado *Arduino programming language* que iremos aprendiendo.

La placa podrá recibir datos del mundo que le rodea y también ofrecer datos a éste. Para obtener información del mundo exterior, Arduino utiliza sus entradas de modo que reciben impulsos eléctricos; mediante diferentes elementos que iremos viendo, podemos observar la realidad que nos rodea para transformarla en corriente eléctrica, a estos elementos comúnmente se les llama sensores. Por ejemplo, podríamos hacer un detector de presencia mediante un botón que cuando algo esté cerca lo apriete; éste no sería el detector más adecuado y es que en cada situación deberemos seleccionar el sensor que más se adapte a las necesidades.

Del mismo modo tenemos las salidas para proporcionar información al mundo exterior; por parte de la placa Arduino se ofrecen unos valores eléctricos que se deben adecuar al receptor de dicha información, para ello se pueden utilizar múltiples métodos, tales como luces, motores, altavoces, pantallas u otras muchas maneras, y será la utilización del montaje la que determine el método de salida, a estos elementos se les suele llamar actuadores. Volviendo al semáforo, podríamos crearlo utilizando tres luces (una roja, una naranja y otra verde) porque es lo que más se adecúa, pero podríamos haber utilizado una pantalla que muestre las palabras “Alto”, “Vaya parando” y “Continúe”... pero ¿verdad que en este caso valdría con mostrar las tres luces?

El hardware ofrecido por Arduino, junto con su entorno de programación, está pensado para ser utilizado por personas sin preparación específica de programación ni electrónica... bueno un poco si se necesita, sobre todo si se quieren hacer montajes complejos, pero veremos que no es necesario tener grandes conocimientos para realizar prototipos realmente interesantes y útiles; lo iremos descubriendo a lo largo del libro y sus ejemplos.

---

**NOTA:** Un pin es una palabra inglesa que significa clavija y es cada uno de los pinchitos o contactos metálicos que existen en los componentes electrónicos. Muchas veces también se les llama así a cada uno los agujeros de los zócalos donde realmente se inserta el pin.

---

# Arduino Uno

Como hemos comentado anteriormente, existen multitud de modelos de placas Arduino, desde algunas muy potentes con muchas funcionalidades, salidas y entradas, hasta otras pequeñas como Arduino Nano especialmente pensada para montajes en los que el espacio es un inconveniente e incluso algunas tan curiosas LilyPad desarrollada para utilizar en ropa y complementos.

Además también hay muchas tarjetas compatibles realizadas por otros fabricantes como Freeduino o la Roboduino y es que tanto los esquemas del hardware Arduino como el código fuente de su software están a disposición de la gente, cualquiera puede acceder a ellos y crear sus propias placas y venderlas.

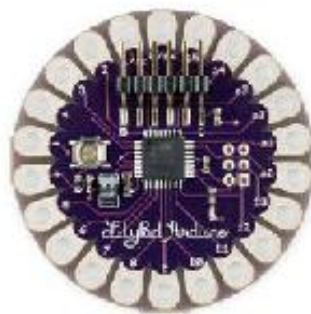


Figura 2.2. Placas Arduino Nano y LilyPad

Nosotros nos centraremos en una de las más extendidas, concretamente la Arduino Uno (muy semejante a la Arduino Duemilanove) aunque si tenemos otra diferente, con muy pocos cambios (como cambiar el número de pin donde realizar las conexiones) podremos adaptar los ejemplos. La última versión de la placa es la Arduino Uno R3 (es decir revisión 3) aunque en nuestro caso no sea de importancia la revisión, ésta se puede constatar en el reverso de la placa (si no aparece es probablemente la R1).

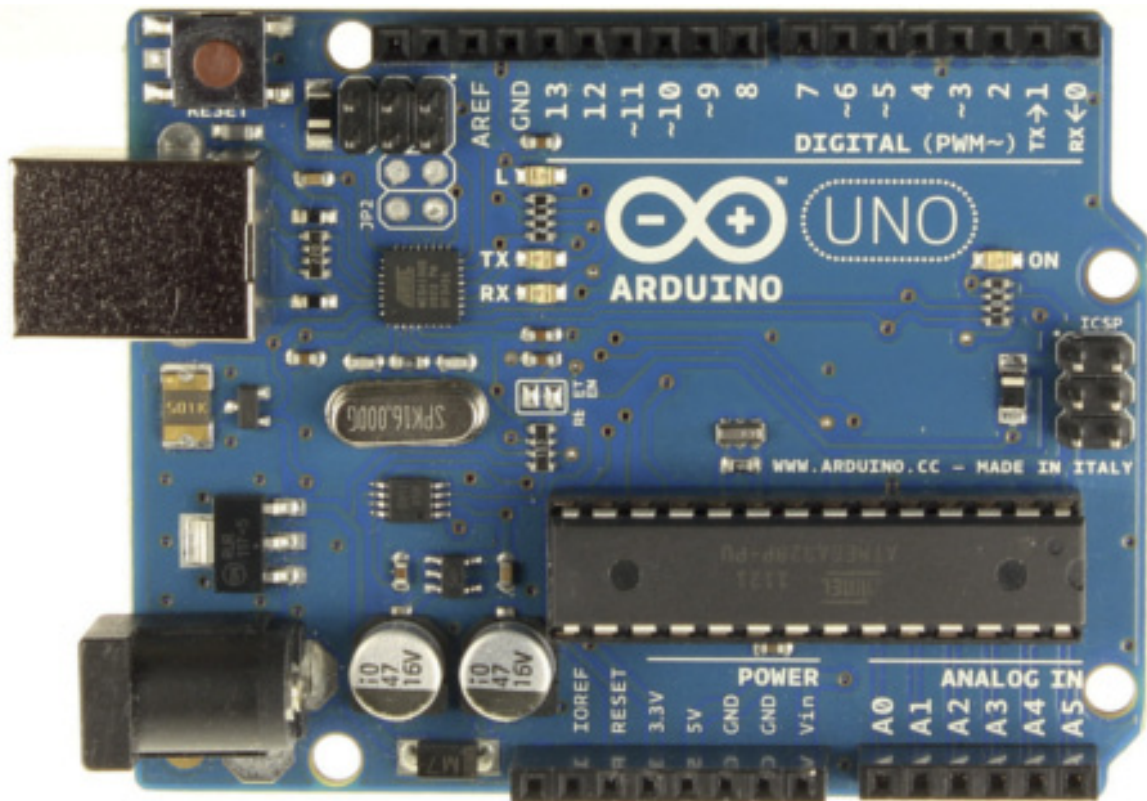


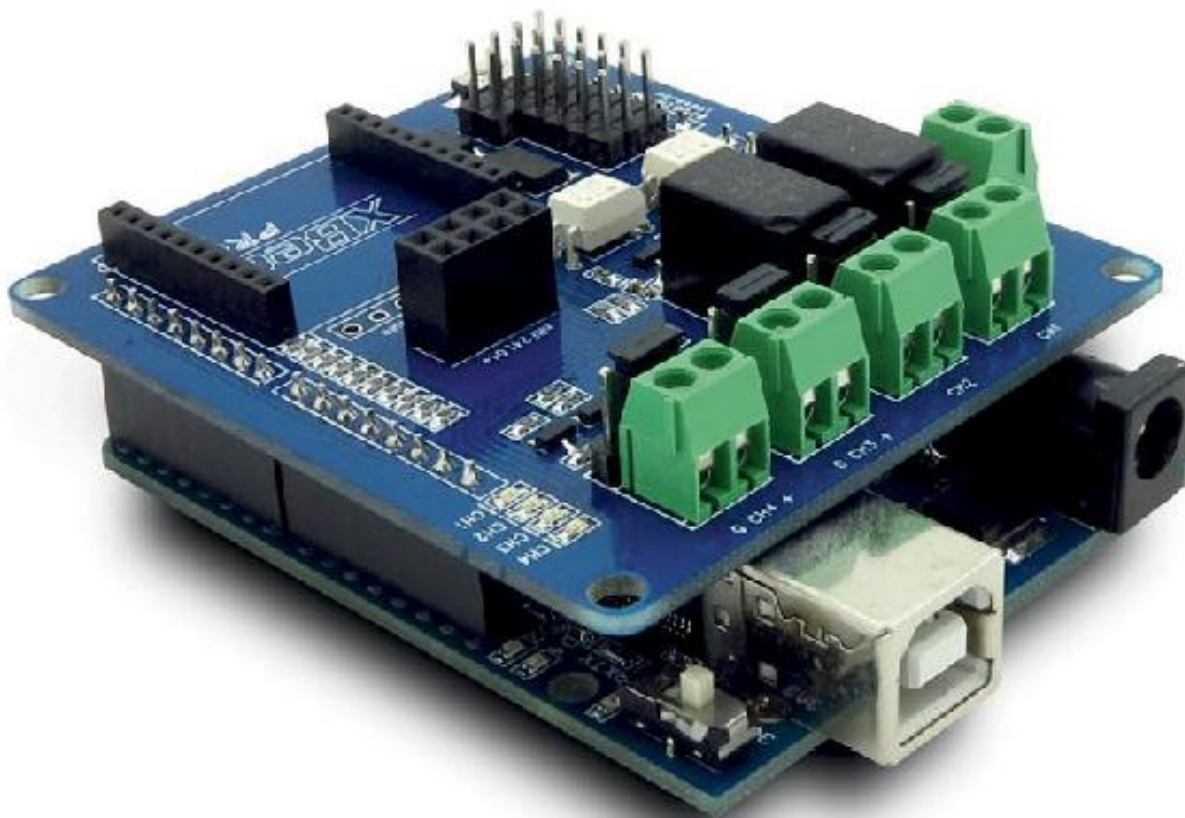
Figura 2.3. Placa Arduino Uno R3

## Shields

Para facilitar los prototipos con piezas requieran conexiones complejas o funcionalidad adicional (como por ejemplo lectores de tarjetas de memoria), los fabricantes ponen a disposición de Arduino unos bloques de conexiones ya montadas a modo de placas que se colocan directamente sobre la tarjeta Arduino, pudiendo usarlas nada más ser colocadas. Estos

bloques son conocidos como *shields* (escudos) y existen de muchos tipos (Ethernet, GPS...).

Normalmente las tarjetas *shield* llevan emparejada una funcionalidad, es decir en la propia placa *shield* van los componentes que ofrecen la funcionalidad, como por ejemplo la *shield* Ethernet lleva el adaptador para la clavija del cable o la de comunicación GSM lleva el habitáculo para introducir la tarjeta de teléfono y los chips necesarios para que funcionen correctamente, pero también existen *shield* de tipo adaptador, que no ofrecen ninguna otra utilidad más que poder conectar otro elemento a ellas, por ejemplo para el uso de pantallas existen adaptadores que facilitan su uso.

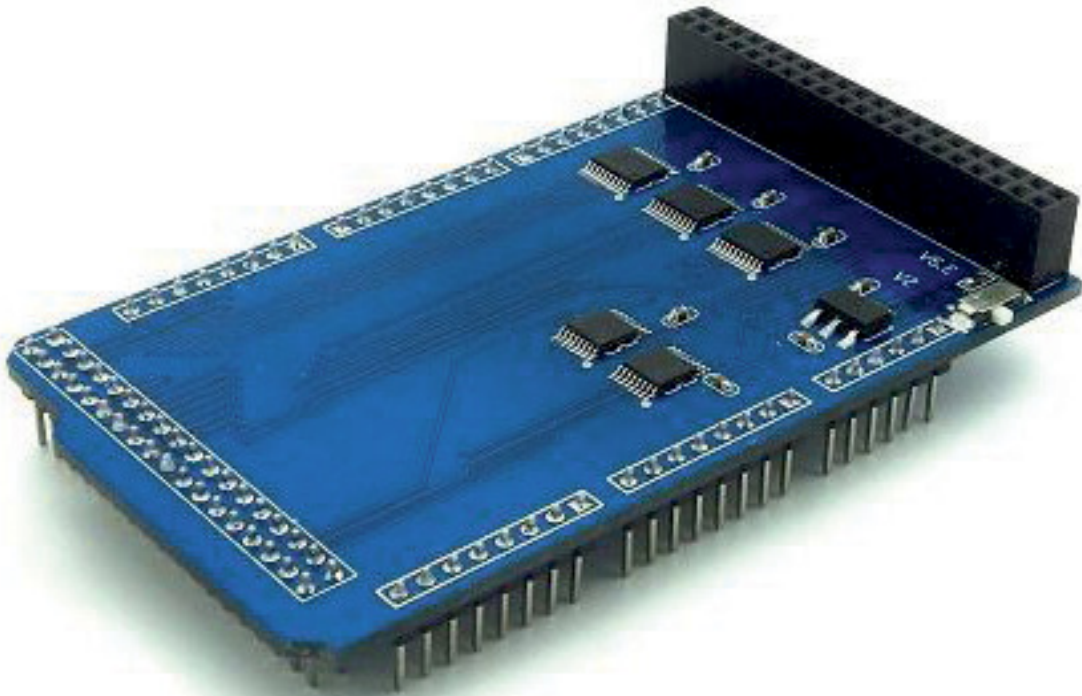


**Figura 2.4.** Placa Arduino con shield montado

Usar las *shield* no es obligatorio, siempre podemos ensamblar los componentes que conforman la *shield* uno a uno por separado hasta



conseguir la configuración necesaria, pero es mucho más laborioso. Por ejemplo si queremos montar una pantalla, usar el adaptador nos ahorraría hasta 40 conexiones, sin tener que preocuparnos de conocer la función de cada uno de ellos.



**Figura 2.5.** Placa *shield* de adaptación para TFT

En caso de utilizar *shields*, otra facilidad que proporcionan es que son fácilmente apilables, pudiendo colocar varias *de ellas* en el mismo montaje, de modo que la señal de la placa Arduino se propaga entre las placas montadas, alimentando los pines necesarios de cada una de ellas.

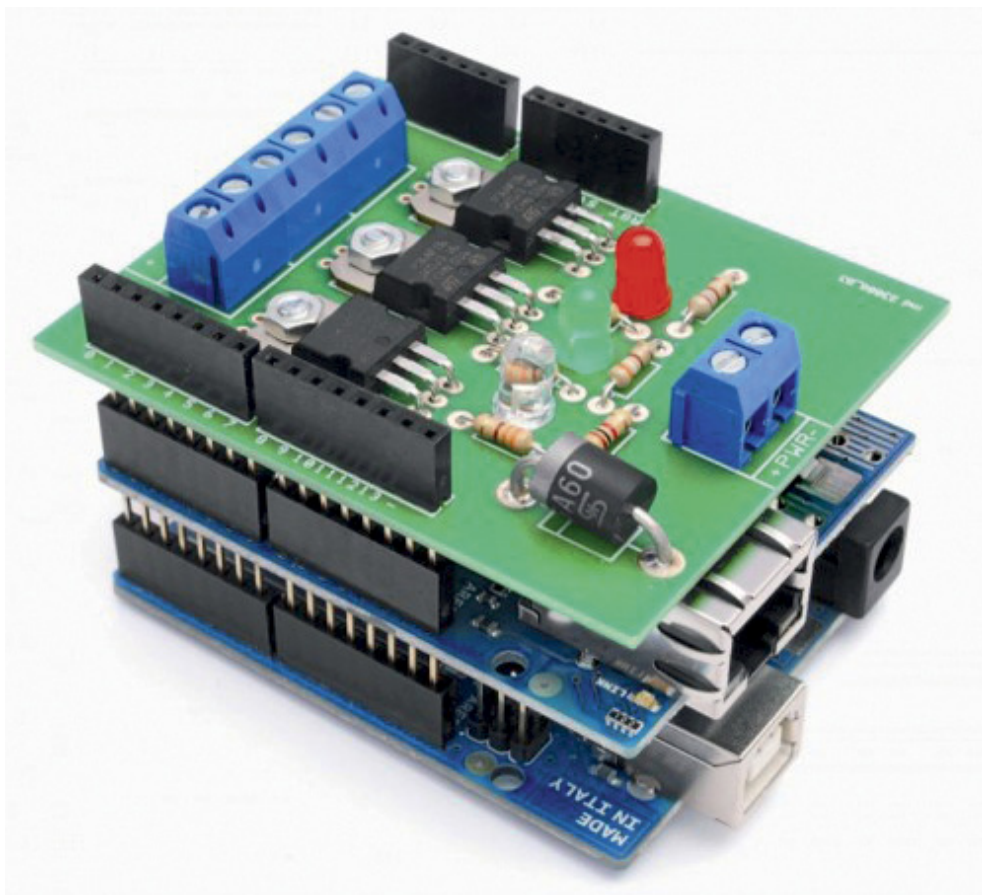
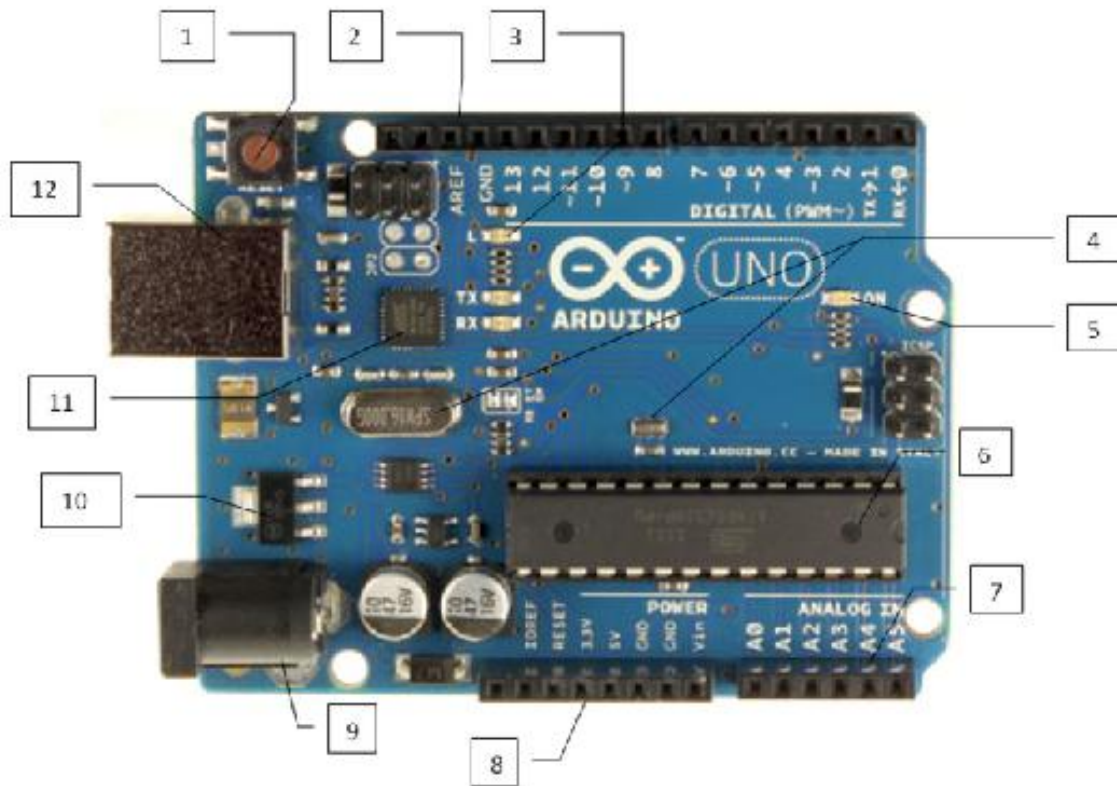


Figura 2.6. Placa Arduino con dos *shield* montados

## Arduino Uno

Vamos a familiarizarnos un poco con el modelo de placa Arduino que vamos a utilizar: la Arduino Uno; la usaremos por ser una de las más fáciles de encontrar (ella y sus copias) y más *shields* tiene disponibles. Su aspecto ya se pudo ver en una figura anterior (y posiblemente lo estemos viendo en vivo en nuestras manos), pero vamos a ver que es cada una de las partes de la placa.



**Figura 2.7.** Elementos de la placa Arduino Uno

1. **Pulsador de Reset:** Sirve para inicializar de nuevo el programa cargado en microcontrolador (el cerebro). Cuando deje de responder la tarjeta, es el botón de apagado/encendido que hace que vuelva a restablecerse.
2. **Puertos de entrada y salida digital:** Son los zócalos donde conectar los sensores y actuadores que necesiten señal digital. Más adelante veremos qué es señal digital y qué es señal analógica.
3. **Led pin 13:** Es un led (una bombilla pequeña) integrado en la placa para poder ser utilizado en montajes. Corresponde al pin 13.
4. **Reloj oscilador:** Es el elemento que hace que la placa vaya ejecutando las instrucciones. Es el reloj que marca el ritmo al cual se debe ejecutar cada instrucción.
5. **Led de encendido:** Es una bombilla pequeña que ilumina cuando la placa está correctamente alimentada.



6. **Microcontrolador:** El cerebro de la placa. Este es el procesador que se encargará de ejecutar las instrucciones que le demos.
7. **Entradas analógicas:** Zócalo con distintos pines de entrada analógica que permiten leer entradas analógicas. Más adelante veremos qué es señal digital y qué es señal analógica.
8. **Zócalo de tensión:** Aquí encontraremos pines con los que alimentar nuestro circuito.
9. **Puerto de corriente continua:** Se utiliza para alimentar de electricidad la placa si no se usa alimentación vía USB.
10. **Regulador de tensión:** Sirve para controlar la cantidad de electricidad que se envía a los pines, asegurando así que no estropeamos lo que se conecte a ellos.
11. **Chip de interface USB:** Es el encargado de controlar la comunicación con el puerto USB.
12. **Puerto USB:** Se utiliza tanto para conectar con el ordenador y transferir los programas al microcontrolador como para dar electricidad a la placa. También se puede usar como puerto de transferencia serie hacia la placa, tanto para transmisión como para recepción.

## Instalación del entorno de programación

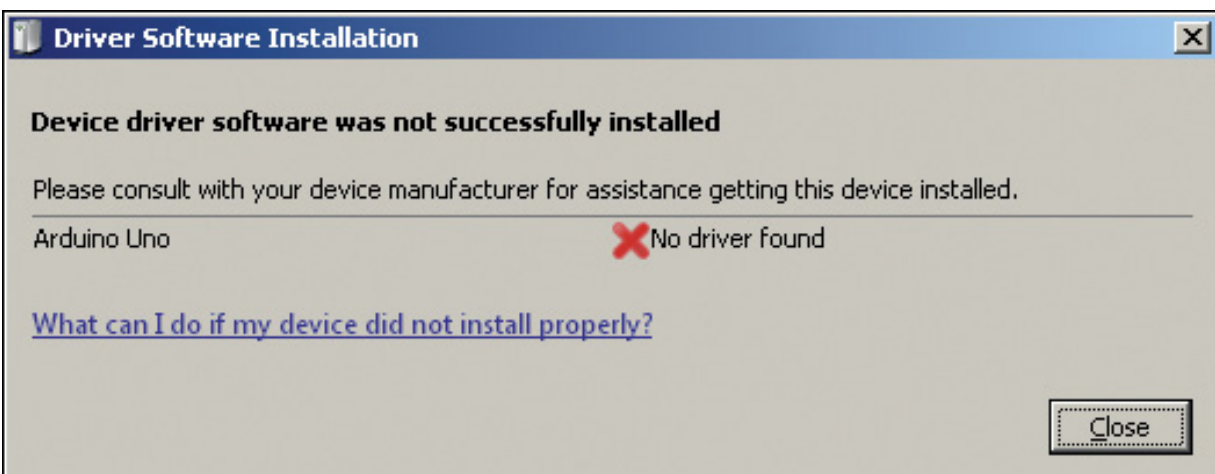
Como se ha comentado anteriormente, las placas Arduino podemos verlas como una especie de PC al que se le tiene que programar para que sea capaz de saber qué debe hacer cuando recibe un impulso eléctrico por alguna de sus entradas o si se desea ofrecer algún dato por alguna de sus salidas. Para poder programarlo se dispone de un entorno de programación que nos facilitará en gran medida el proceso de desarrollo del código. Este entorno de programación está disponible para Windows, Mac y Linux; y se puede descargar de manera gratuita desde <http://arduino.cc/en/Main/Software>. En el momento en el que se realiza este libro, la versión estable es la 1.6.3.

Podemos optar por el fichero de instalación o el fichero zip. Si se descarga el archivo zip, no necesitaremos permisos especiales para instalarlo en el ordenador; este fichero contiene todo lo necesario para la ejecución del entorno de programación y no es necesaria su instalación, simplemente vale con descomprimirlo.

Cuando se descomprime el fichero, se obtienen una serie de directorios que no veremos su contenido, simplemente quedarnos con el directorio principal donde está el ejecutable y el directorio `drivers` que necesitaremos para que nuestra placa y el ordenador puedan conectarse.

Antes de poder trabajar con la tarjeta Arduino, y dependiendo del sistema operativo que tengamos, puede ser necesario instalar ciertos drivers. Dependiendo del sistema operativo (la mayoría será Windows) que se vaya a utilizar, nos habremos descargado un fichero u otro del entorno de desarrollo y dependiendo de éste, el directorio `drivers` contendrá unos elementos u otros específicos al sistema operativo.

Las instalaciones para sistemas operativos Mac y Linux son muy semejantes y básicamente es descomprimir el fichero y ponerse a trabajar, pero en caso de utilizar Windows el proceso es un poco más largo. Si conectamos la placa Arduino al ordenador mediante el cable USB, saltará un mensaje conforme no se tienen instalados los drivers para el elemento USB conectado, intentará buscarlos e instalarlos pero seguramente fallará.



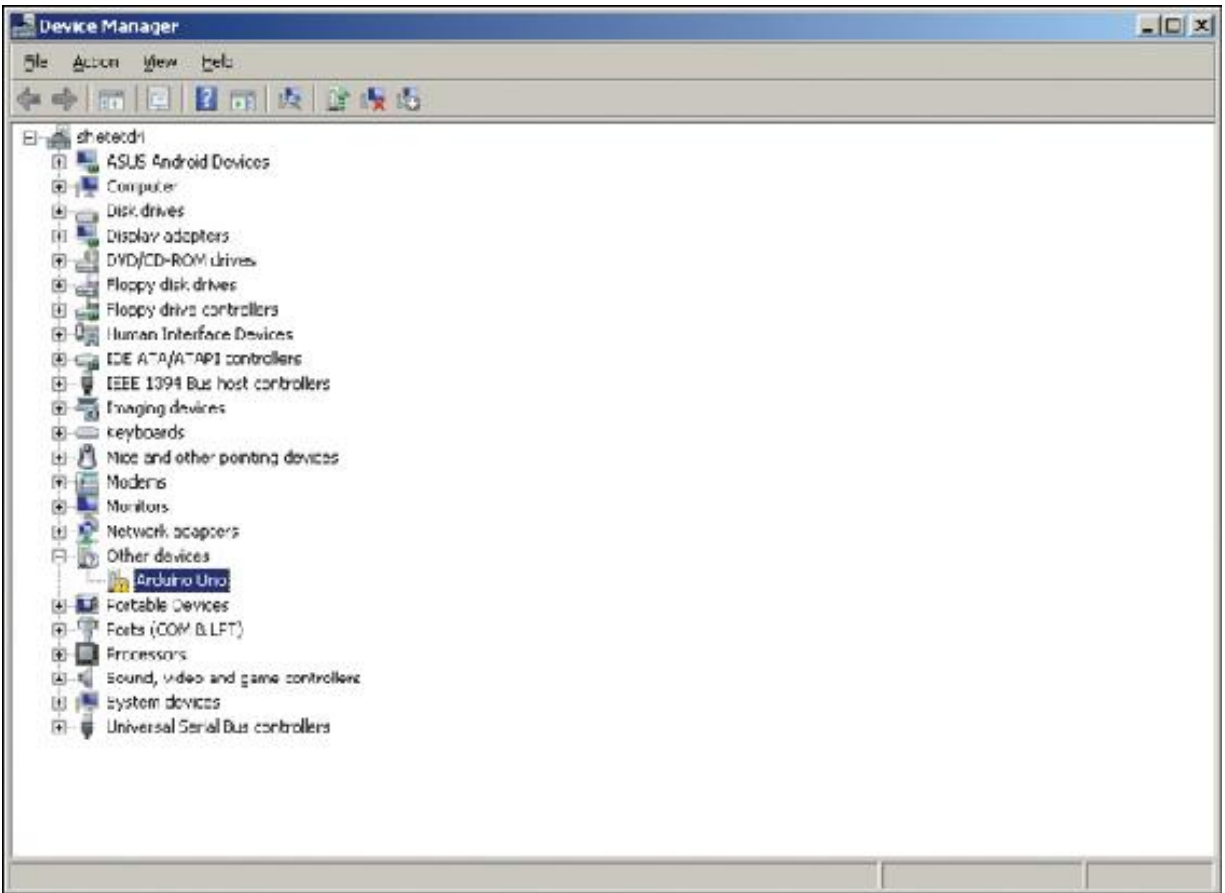
**Figura 2.8.** Error al instalar los drivers Arduino Uno

Para solventar este problema hay que dirigirse al gestor de dispositivos de Windows, para ello se pueden seguir varios caminos, entre otros:

- Pulsar con el botón derecho sobre **Mi PC** y seleccionar **Propiedades**, una vez en la nueva pantalla ya puede seleccionarse el **Gestor de Dispositivos**.
- Ir a **Panel de Control** seleccionar la opción de **Seguridad y sistema**, nuevamente seleccionar **Sistema** y ya se tendrá acceso al **Gestor de Dispositivos**.

Dentro del Gestor de dispositivos, se puede ver que hay un error en la instalación del Arduino Uno (figura 2.9).

Si se pulsa mediante el botón derecho sobre la entrada correspondiente al error y se selecciona la entrada **Propiedades**, aparece una pantalla con información sobre el dispositivo y un botón para actualizar el driver (figura 2.10).



**Figura 2.9.** Error en el Gestor de Dispositivos

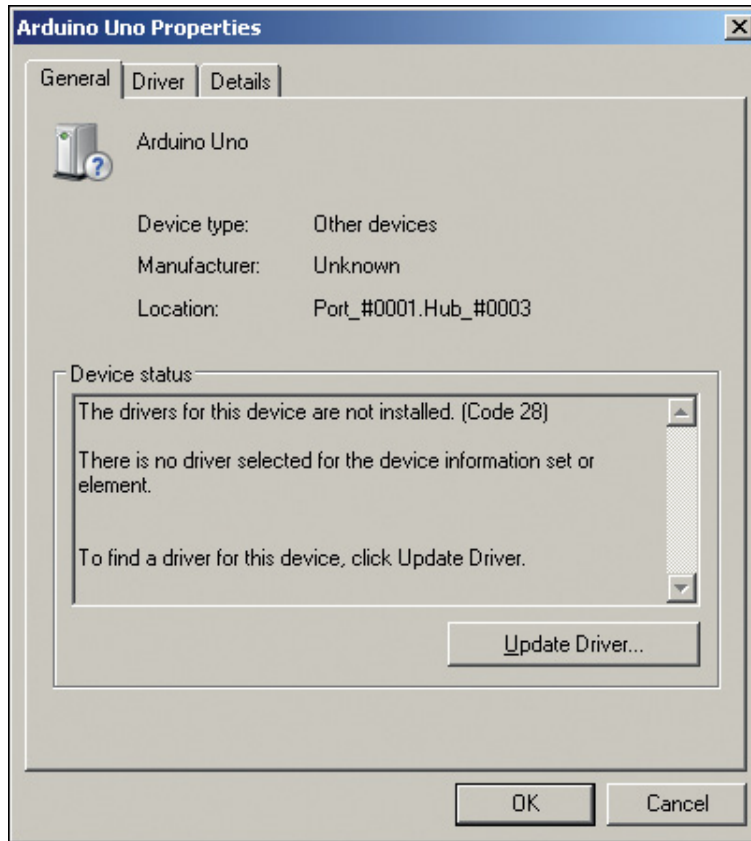


Figura 2.10. Actualización del driver para Arduino Uno

Seleccione el directorio drivers correspondiente al fichero descargado que se ha descomprimido y Windows comenzará la instalación del driver adecuado.

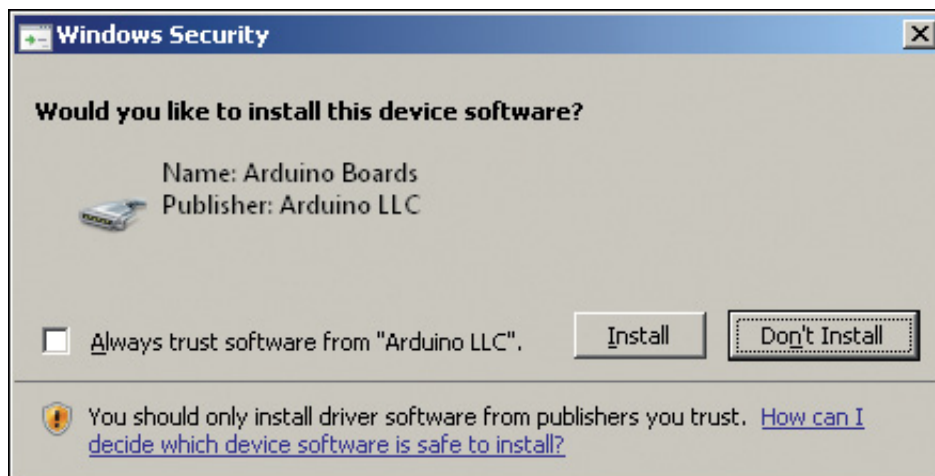


Figura 2.11. Driver encontrado, se procede a la instalación

Estos drivers harán que el puerto USB aparezca como un puerto COM (serie) dentro del PC, lo que facilitará la comunicación con el dispositivo Arduino conectado a ese puerto. En caso de que fuera necesario, se pueden obtener más drivers y actualizaciones en <http://www.ftdichip.com/Drivers/VCP.htm>. Muy importante es también apuntarse el número del puerto COM que usará el ordenador, ya que lo usaremos más adelante. En el caso de la figura 2.12 es el COM27.

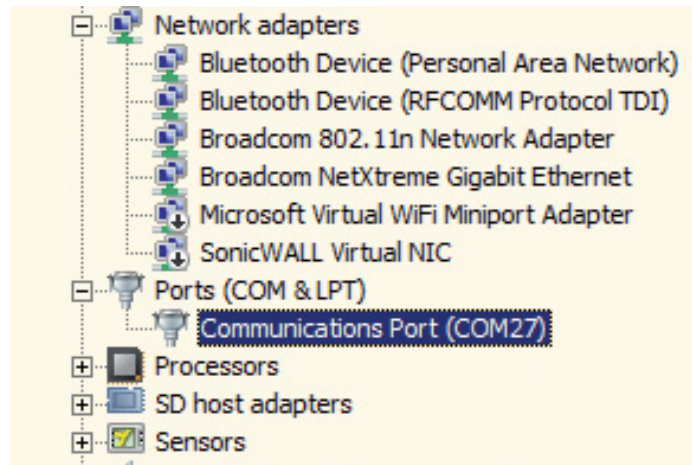
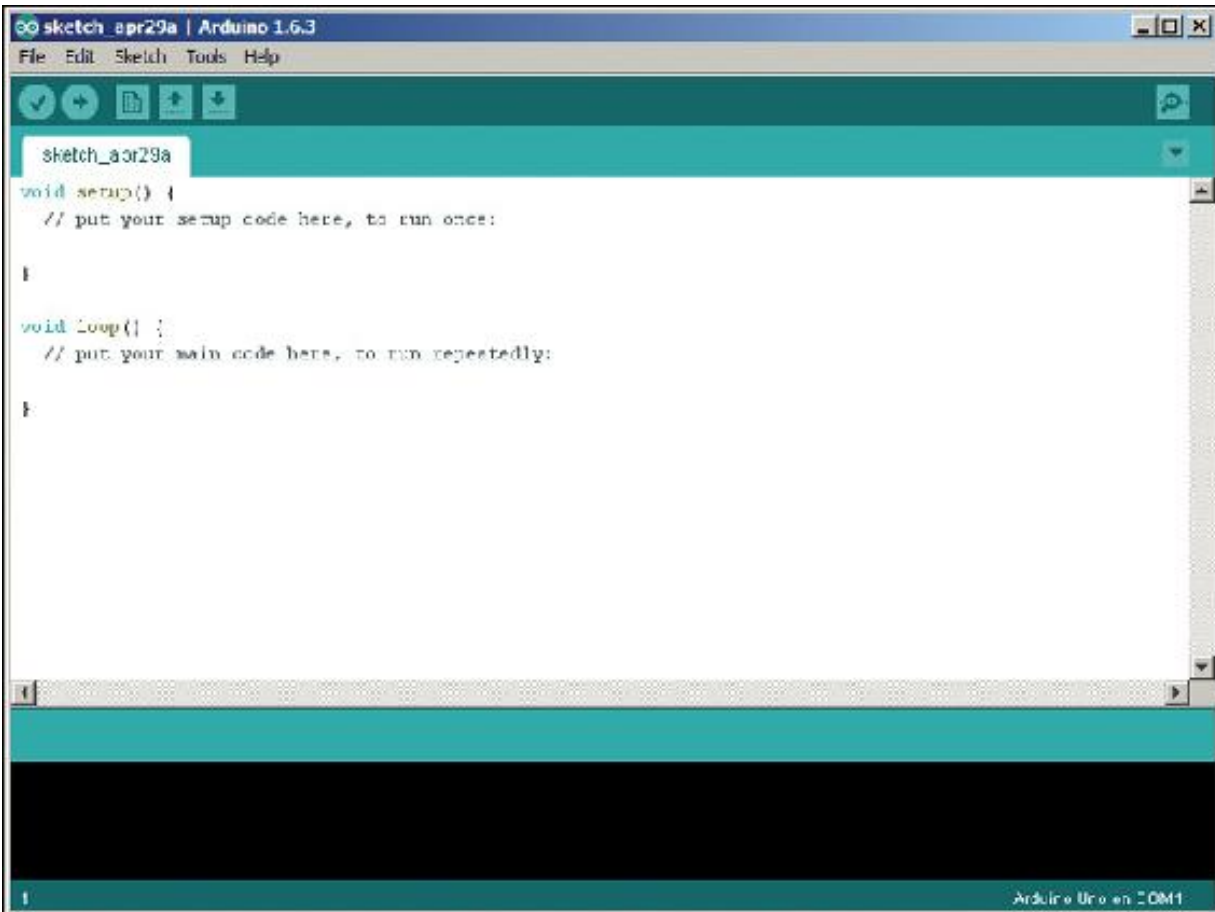


Figura 2.12. Instalación del driver completada

## Entorno de programación

Para ejecutar el entorno de programación hay que seleccionar el fichero **arduino.exe** (o **arduino** en Linux y Mac) dentro del directorio de donde se ha descomprimido el fichero `.zip` o se ha hecho la instalación. El aspecto del entorno de desarrollo es el de la figura 2.13, aunque dependiendo del sistema operativo que se esté utilizando puede haber ligeras diferencias, el aspecto general es muy similar.



**Figura 2.13.** Entorno de desarrollo de Arduino

Podemos ver que la ventana del entorno de desarrollo está dividida en tres grandes bloques horizontales (sin contar los menús, ya que en Mac no se encuentran en la ventana); la parte superior corresponde a una botonera con las acciones más comunes, la parte central será donde realicemos el trabajo de programación, donde daremos las instrucciones a la tarjeta; el conjunto de órdenes que se escriben se llama *sketch*; por último, la parte inferior corresponde a la salida de la consola, donde podremos ver errores y mensajes de información. En la parte de abajo de esta consola se encuentra una barra de información donde se puede el número de línea en la que está posicionado el cursor dentro del *sketch* y a la derecha el modelo de placa Arduino activa en ese momento y puerto (el “enchufe” del ordenador) en el que se espera que esté conectada.



**Figura 2.14.** Botonera del entorno de desarrollo de Arduino

---

**NOTA:** En el mundo Arduino, cada uno de los programas ejecutables por las placas son denominados *Sketch*.

---

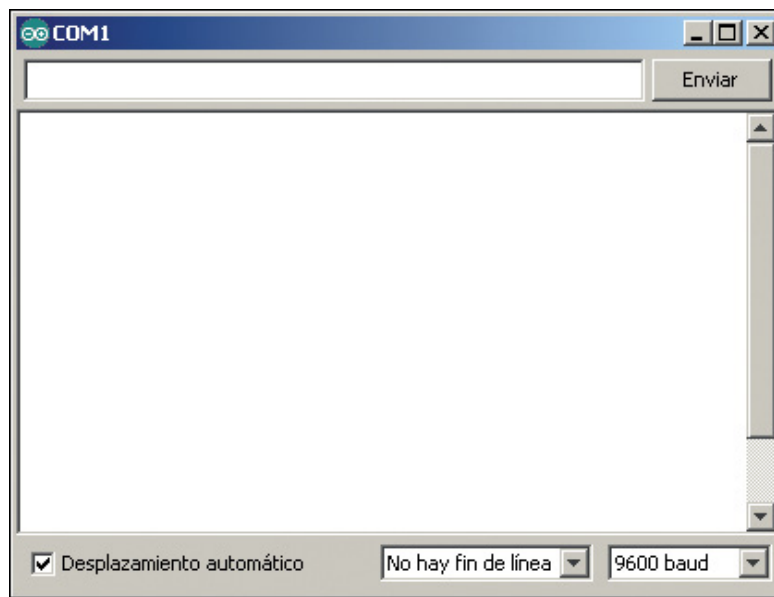
De izquierda a derecha la función de los botones es la siguiente:

- **Verificar:** Este botón verifica que el código fuente (lo que hemos escrito en el bloque del medio, a partir de ahora lo llamaremos *sketch*) es correcto.
- **Cargar:** Transmite el programa a la placa Arduino que esté conectada al ordenador en ese momento, a partir de ese momento la placa comenzaría a trabajar con nuestro programa; sería semejante al botón ejecutar de otros entornos de programación. Hay que asegurarse que tanto el modelo de la placa como el puerto al que está conectada se han configurado correctamente, más adelante se verá donde realizar esta configuración. Es muy recomendable salvar el *sketch* antes de cargarlo en la placa, ya que podría quedarse colgado el sistema y perder el trabajo. También es recomendable verificar el código antes de cada carga con tal de evitar problemas posteriores.
- **Nuevo:** Crea un nuevo *sketch*. En caso de haber modificado el actual pide que se guarde.
- **Abrir:** Muestra una selección de *sketches* correspondientes a ejemplos disponibles con el entorno y *sketches* propios disponibles en el *sketchbook* (repositorio de *sketches*).
- **Guardar:** Este botón guarda el trabajo realizado. Los ficheros son guardados por defecto en el directorio de documentos del usuario, dentro de la carpeta `Arduino`. Dentro de esta carpeta encontraremos una nueva carpeta por cada *sketch* que hayamos realizado. Dentro de las carpetas correspondientes a los *sketches*, se encuentran los ficheros con el código fuente, que tienen por defecto la extensión. Se



guardará un archivo *.ino* por cada pestaña de código que tengamos abierta.

- **Monitor serie:** Por último y situado a la derecha del todo de la barra de herramientas, se encuentra el botón del monitor serie. Esta pantalla o monitor es una herramienta muy importante sobre todo cuando se trata de encontrar problemas en el programa. El monitor muestra los datos enviados desde Arduino y también permite el envío de datos hacia la placa. Pulsando sobre el botón se abre una nueva ventana como la que se muestra en la figura 2.15.



**Figura 2.15.** Monitor serie

En el monitor serie se puede observar una caja de texto en la parte superior; este campo permite introducir datos para enviar hacia la placa Arduino, por ejemplo simular que tenemos un teclado conectado en la placa y enviarle las pulsaciones de las teclas de éste; el envío se realiza al pulsar el botón situado a la derecha o al pulsar la tecla **Enter**. Debajo se encuentra un área blanca que será donde se muestren los mensajes enviados desde Arduino. En la parte de abajo de la pantalla podemos ver una casilla de selección que hará que el área de salida de mensajes se vaya desplazando conforme salgan nuevos mensajes o se quede quieta; encontramos también dos cajas desplegables, la primera de ellas permite

controlar el comportamiento de nueva línea y retorno de carro en los mensajes y la segunda se encarga de configurar la velocidad de comunicación con la placa Arduino medido en baudios (*baud*), el *baud* es el número de cambios del estado de la información por segundo, es decir 9600 *baud* significa que en cada segundo se transmitirán 9600 caracteres.

Por defecto, los *sketch* no envían ningún tipo de dato al monitor serie, sino que debe hacerse mediante programación, indicando qué, cómo y cuándo enviarlo. Del mismo modo, tampoco se recibe ningún dato si no se especifica lo contrario mediante código.

A lo largo del libro se experimentará con el monitor, tanto para encontrar errores en los programas y montajes como para introducir datos hacia la placa Arduino.

En cuanto a la zona de menús vamos a analizar las entradas más importantes, no hay que memorizarlas, de hecho no usaremos todas, pero por si alguna vez nos entra la duda de para qué sirve, pues al menos tener esta pequeña guía o referencia.

Dentro del menú Archivo (`File`) podemos encontrar las siguientes entradas:

- **Nuevo (New):** Permite crear un nuevo *sketch* en blanco.
- **Abrir (Open):** Sirve para abrir *sketches* ya existentes. Genera una nueva ventana con el *sketch* seleccionado.
- **Sketchbook:** Es la librería de *sketches* guardados.
- **Ejemplos (Examples):** Contiene una serie de ejemplos de *sketches* operativos que podemos utilizar en nuestros trabajos. Se presentan convenientemente ordenados por tipo.
- **Cerrar (Close):** Cierra el *sketch* actual.
- **Guardar (Save):** Guarda el *sketch* activo.
- **Cargar (Upload):** Carga el *sketch* actual en la tarjeta configurada que esté conectada en el puerto configurado.
- **Cargar usando programador (Upload Using Programmer):** Carga el *sketch* en el microcontrolador utilizando un programador en lugar

de la tarjeta Arduino. Esta opción no se usará en el libro.

- **Preferencias (Preferences):** Nos guía a la pantalla de preferencias donde configurar el comportamiento del editor. Más adelante se explica en detalle.

En el menú Editar (`Edit`) encontramos:

- **Copiar, Pegar...(Copy, Paste...):** Dan acceso a funcionalidades típicas de cualquier editor de textos.
- **Copiar como HTML (Copy as HTML):** Copia el código del *sketch* en formato HTML para poderse insertar en páginas web.
- **Copiar para el Foro (Copy for Forum):** Copia el código del *sketch* en un formato especial para utilizar en el foro de Arduino, incluyendo colores dependientes de sintaxis (podemos acceder al foro mediante la URL <http://arduino.cc/forum/index.php?action=forum>).
- **Comentar/Descomentar (Comment/Uncomment):** Permite comentar y descomentar bloques de código.
- **Buscar (Search):** Varias entradas de menús con toda la familia de posibilidades típicas de búsqueda de los editores de código.

El menú Sketch tiene entre sus opciones:

- **Verificar/compilar (Verify/compile):** Procede a verificar la sintaxis del programa, es idéntico a la acción realizada por el botón comentado anteriormente.
- **Mostrar la carpeta de Sketch (Show Sketch Folder):** Abre el explorador de sistema en la carpeta que contiene el *sketch* actual.
- **Agregar Archivo (Add File):** Permite añadir nuevos archivos de código al *sketch* activo en ese momento.
- **Importar Librería (Include Library):** Añade al código las líneas necesarias para utilizar la librería seleccionada (elementos `#include`). Las librerías disponibles son las que se encuentran almacenadas en el disco duro, dentro del directorio *libraries*. El uso

de las librerías permite la reutilización de código realizado por nosotros mismos o por un tercero y que en algunos casos facilitará de sobre manera el uso de los ciertos componentes, como por ejemplo la comunicación con Internet desde la placa Arduino.

Dentro del menú Herramientas (Tools) encontramos:

- **Formato automático (Auto Format):** Organiza y ordena el código de modo que quede perfectamente tabulado y fácil de leer.
- **Archivar el Sketch (Archive Sketch):** Crea un fichero comprimido con todo el código fuente correspondiente al *sketch*. Ideal para portar los *sketches* entre distintas máquinas o realizar backups.
- **Monitor serie (Serial Monitor):** Muestra el monitor serie del mismo modo que el botón comentado anteriormente.
- **Tarjeta (Board):** Muestra una serie de tarjetas entre las cuales tenemos que seleccionar la que se quiere utilizar para probar el *sketch*.
- **Puerto serie (Port):** Muestra los puertos serie disponibles. Se debe seleccionar aquél en el que se encuentre conectada la tarjeta sobre la que se quiere trabajar.
- **Programador (Programmer):** En caso de no utilizar las tarjetas Arduino y programar directamente el microcontrolador, se debe seleccionar el programador que se va a utilizar mediante este menú. En este libro no se usará esta opción.
- **Grabar secuencia de inicio (Burn Bootloader):** Permite la grabación del *Bootloader* de Arduino en el chip. El *Bootloader* es una pieza de código que hace que el chip sea compatible con Arduino). La placa Arduino permite extraer el chip microcontrolador y remplazarlo por uno nuevo, esto da la opción de utilizar la misma placa con distintos chips donde podemos tener cargados diferentes programas o usar los chips en otros proyectos con placas propias. A la hora de comprar los chips ATmega podemos comprarlos con el *Bootloader* de Arduino ya cargado o no. Lo más sencillo es comprarlos con el *Bootloader* ya grabado, pero en caso

de no hacerlo, esta sería la entrada a utilizar para hacerlos compatibles con Arduino.

Por último en el menú de Ayuda (`Help`) podemos encontrar varias entradas con referencias a ayudas y más información sobre el entorno de programación o el mundo Arduino en general.

La configuración del entorno se realiza como se explicaba unas líneas atrás, mediante el menú Archivo>Preferencias, que nos dirige a la ventana de preferencias. En esta nueva ventana se pueden configurar valores que afectan al aspecto del entorno de desarrollo como el tamaño de letra a utilizar o el idioma y otros valores que afectan más de modo funcional, como la asociación de la extensión `.ino` este programa o el directorio de trabajo.

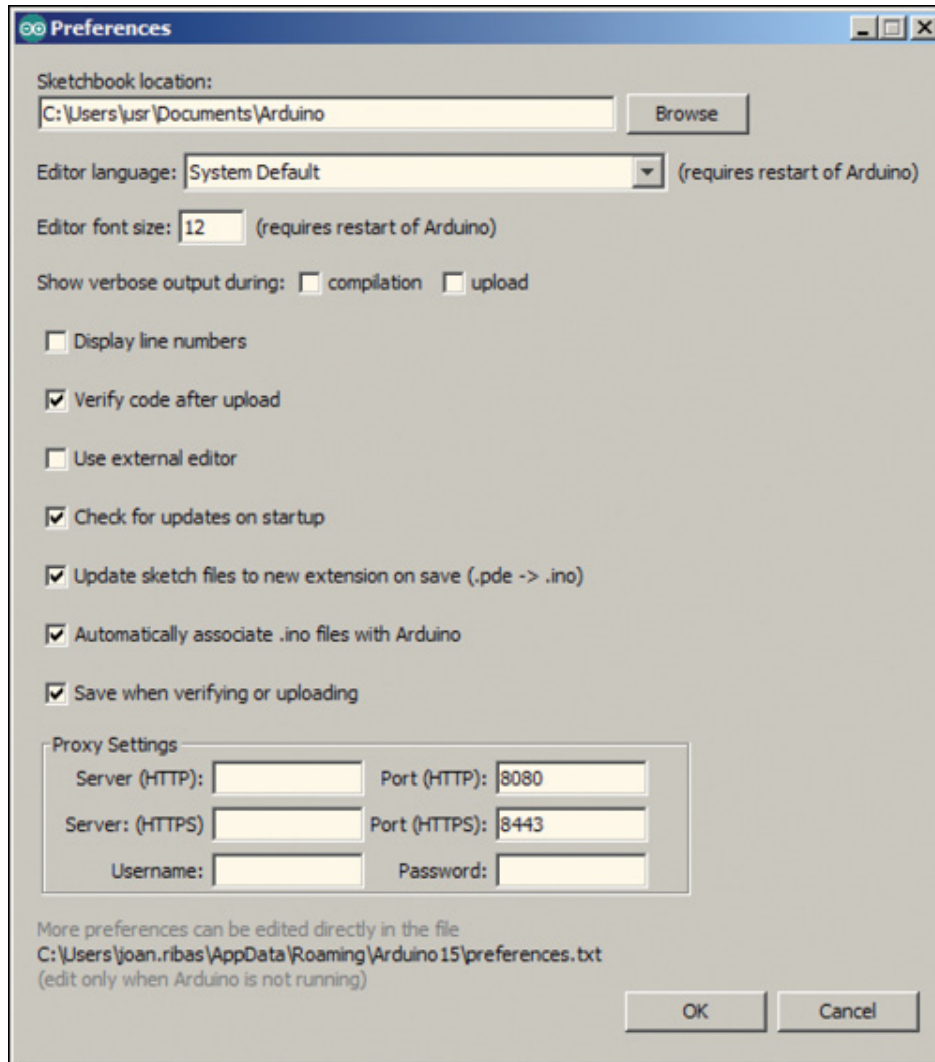


Figura 2.16. Ventana de preferencias

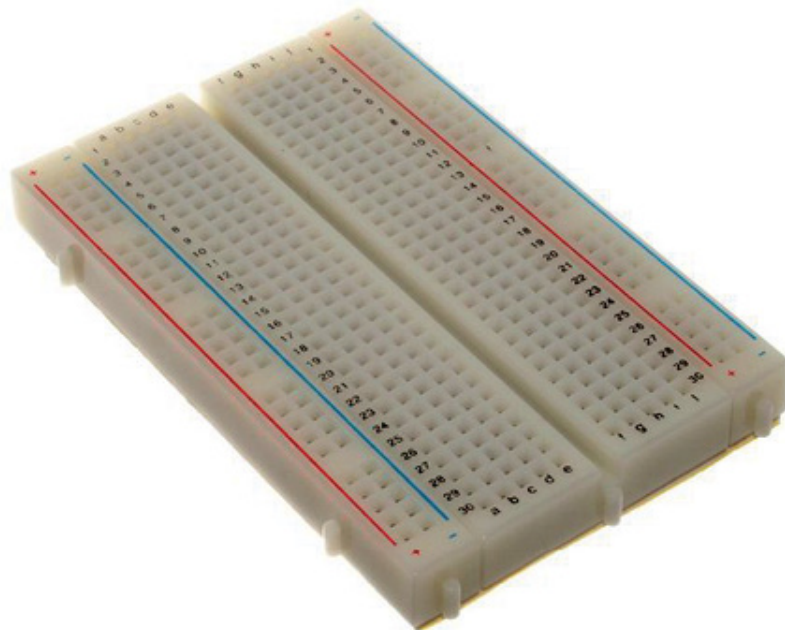
En la parte central es donde daremos las órdenes, o dicho de otro modo, donde codificaremos los programas o *sketches*. El entorno de Arduino ofrece algunas ayudas como colores en el texto para la sintaxis de programación.

## Protoboard o breadboard

La placa Arduino está pensada para interactuar con el mundo real y esta interacción la realiza través de sensores para obtener información del

mundo exterior (por ejemplo medir la humedad) y de actuadores para modificarlo (por ejemplo abriendo una válvula que expulse vapor).

Alguno de los modelos de placas Arduino disponen de algunos sensores y actuadores integrados, pero lo normal es trabajar tanto con sensores como con actuadores externos. En el montaje final, estos elementos irán fijados a una placa de circuito y las conexiones entre ellos se realizarán con cobre “pegado” a dicha placa, pero hasta ese montaje final, es posible que haya que hacer y deshacer conexiones, con lo que trabajar con placas de circuito impreso es totalmente ineficiente. Para ayudarnos en la preparación del circuito usado para pruebas, utilizaremos una placa de prototipos también conocida como *protoboard*, *plugboard* o *breadboard*.



**Figura 2.17.** Protoboard o breadboard

Dependiendo del modelo, el *protoboard* puede ser de mayor o menor tamaño o tener dibujados unos símbolos u otros, pero su uso es similar. Se trata de una placa agujereada sobre la cual se insertarán los pines de los componentes y extremos de los cables para simular el circuito impreso. Ciertos agujeros están conectados entre sí por dentro de la placa y permiten crear conexiones con menos elementos. Vamos a hablar ahora un

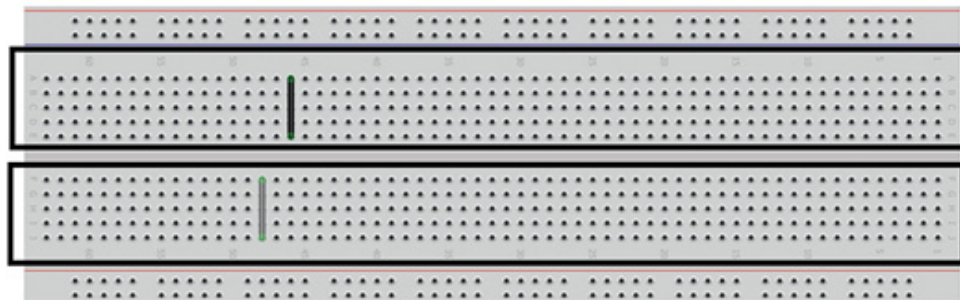
poco sobre cargas y polaridades; si no se entiende, no hay que preocuparse, que en el siguiente capítulo se explicará con más detalle.

Puesta en horizontal, el *protoboard* tiene una o dos líneas de alimentación en los bordes superior e inferior, normalmente marcados con los signos + y - para notar las polaridades (las cargas) de los mismos (realmente la polaridad se la damos cuando conectemos los cables). Todos los agujeros correspondientes a cada línea de alimentación se hayan conectados entre sí horizontalmente, es decir si “hay electricidad” en uno de ellos, esta existirá en todos los agujeros de la línea en igual cantidad. En los ejemplos del libro se utilizará el polo positivo para como línea de alimentación y el negativo como línea de tierra o carga 0.



**Figura 2.18.** Protoboard con dos zonas de alimentación

Por otro lado se tienen los agujeros interiores, que en las placas más habituales se encuentran divididos en dos bloques. Estos agujeros están conectados internamente entre ellos de modo vertical, de forma que todos los agujeros correspondientes a un bloque y de modo vertical, serían el mismo punto dentro de un circuito, o dicho de otro modo, tendrían “la misma cantidad de electricidad” (la misma tensión). Los bloques están separados entre sí eléctricamente, están aislados entre ellos.





**Figura 2.19.** Zona de montaje en la protoboard

Las *protoboard* suelen venir acompañadas de unas letras y números de modo que cada orificio de la placa puede identificarse por la pareja de letra y número. En el caso de la placa expuesta en el dibujo, puesta en vertical (que es como se leen los números), existen sesenta y tres filas de conexiones en dos bloques independientes de cinco columnas cada uno con sus correspondientes letras.

En caso de ser un proyecto muy complejo es posible que se necesiten varias *protoboard*; para ello suelen tener unos enganches en los laterales que facilitan el ensamblado entre ellas.

# 3

## Electricidad y electrónica

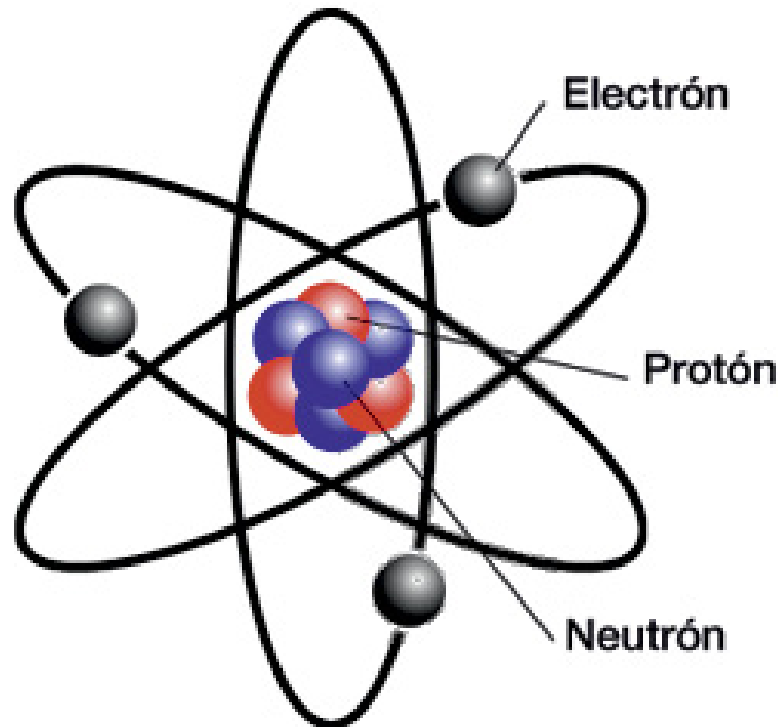


**En este capítulo aprenderá a:**

- Conocer los fundamentos de la electricidad
- Diferenciar electrónica digital y analógica
- Conocer la numeración binaria
- Realizar operaciones binarias

## **¿Qué es la electricidad?**

La electricidad es aquello que hace que nuestra televisión funcione, que la radio suene y que la bombilla se ilumine; es una forma de energía que puede hacer que las cosas se muevan, se calienten y funcionen. Para explicar bien la electricidad tenemos que ver cómo están formadas las cosas de nuestro alrededor, a nivel muy pequeño (no visible a simple vista). Todo está formado por unas partículas que se llaman átomos que están a su vez formados por un núcleo y unos electrones que dan vueltas a su alrededor (como el sol y los planetas). Los electrones que están girando en torno al núcleo tienen una propiedad que se llama carga, y en su caso es negativa mientras que en el núcleo se encuentran los protones con carga positiva y neutrones con carga neutra. Del mismo modo que con los imanes, las partículas con misma carga se repelen mientras que con distinta carga se atraen.



**Figura 3.1.** Átomo

La electricidad es el movimiento de electrones de un punto a otro. Los electrones llevan consigo la carga y ésta es utilizada por ejemplo para encender una bombilla, el televisor o el ordenador. Todo funciona bajo el mismo principio: el movimiento de electrones.

Cuando hablamos de electricidad, un concepto muy común es el de circuito; un circuito es un lazo cerrado que permite que los electrones (la carga) se desplacen de un punto a otro, así cuando se dice que el circuito está abierto, puede haber carga pero no hay electricidad que fluya, del mismo modo que cuando está el circuito cerrado, entonces la electricidad sí fluye. Es como tener un grifo donde puede haber agua pero ésta no sale hasta que no se abra el grifo.

Por ejemplo en la figura 3.2 podemos ver un circuito con una pila, un interruptor y una bombilla. En este circuito existe una carga en la pila, pero esta carga no comenzará a desplazarse hasta que no se cierre el circuito, momento en el que se encenderá la bombilla. Esto no quiere decir que si vemos un cable lo podamos tocar sin más porque “el circuito está

abierto” y no pasará la electricidad; ya que hay otros conceptos que no vamos a entrar en ellos que harían que nuestro cuerpo cerrara el circuito y entonces la carga pasaría por nosotros y dependiendo de la cantidad de carga, las consecuencias serían distintas. Recordad que no hay que perderle nunca el respeto a la electricidad.

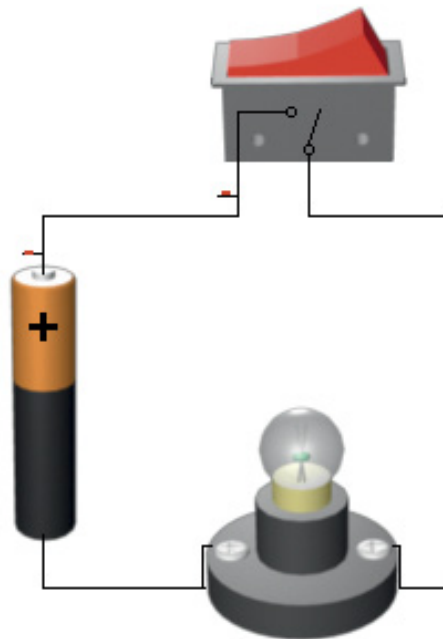


Figura 3.2. Circuito con bombilla

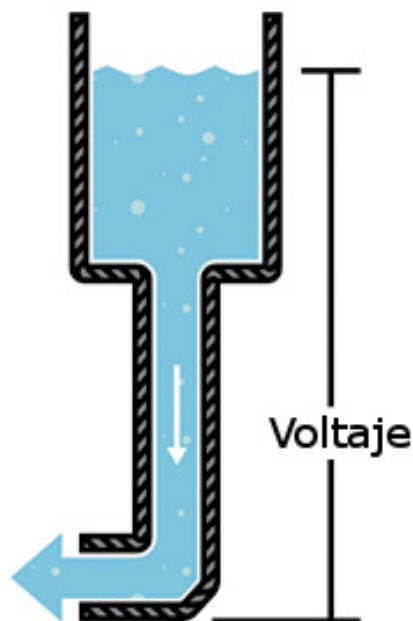
Para ir adentrándonos en los circuitos eléctricos, vamos a ver los tres pilares básicos de éstos que son:

- **Voltaje:** Diferencia de carga entre dos puntos.
- **Corriente:** La velocidad o ritmo con el que fluye la carga.
- **Resistencia:** La tendencia a oponerse a que fluya la carga.

Como puede resultar un poco complicado entender estos conceptos, vamos a intentar compararlo con unos tanques de agua.

## Voltaje

Es la diferencia de carga o diferencia de potencial entre dos puntos de un circuito. Un punto tiene un nivel de carga A, otro punto tiene otro nivel de carga B y su diferencia es el voltaje que existe entre los dos puntos es decir A-B (que puede ser positiva o negativa). La unidad es el Voltio en honor al físico Alessandro Volta (inventor de la batería química) y se representa por la letra V.



**Figura 3.3.** Equivalencia del voltaje con el agua

Para explicar el voltaje mediante los tanques de agua, diríamos que la carga es el volumen de agua y el voltaje es la presión ejercida por el agua.

Entonces, si consideramos un tanque con agua del cual sale una manguera, podemos decir que la presión al final de la manguera sería el voltaje y el agua en el tanque representaría la carga. Entonces, cuanto más agua haya en el tanque, más carga habrá y por lo tanto más presión se medirá al final de la manguera.

El tanque nos sirve también de comparación para las baterías o pilas, ya que es un lugar donde almacenamos cierta cantidad de energía, de carga, que podemos liberar. Según vamos liberando el agua del tanque, la presión al final de la manguera disminuye, del mismo modo que cuando usamos mucho unas pilas, su voltaje disminuye también. Otro efecto que tenemos

al liberar el agua por la manguera es que el flujo a través de ella, también va decreciendo, lo cual nos lleva al concepto de corriente.

## **Corriente o intensidad**

La corriente sería la cantidad de agua que fluye desde el tanque a través de la manguera. Cuanta más presión, mayor será la cantidad de agua que fluya y viceversa. En electricidad mediremos la cantidad de carga que fluye en una cantidad de tiempo, del mismo modo que con el agua mediríamos la cantidad de agua que fluye por la manguera en un tiempo determinado.

La corriente se mide en amperios cuya unidad se representa con la letra A y es en honor al matemático y físico André Ampere. La corriente se representa por la letra I.

Volviendo al ejemplo de los tanques con agua, si suponemos dos tanques cada uno con su correspondiente manguera con la misma cantidad de agua en su interior, pero uno de ellos tiene la manguera más estrecha que el otro, mediremos la misma presión al final de cada una de las mangueras, pero en cuanto el agua comience a fluir, lo harán de manera distinta, de modo que el que tenga la manguera más estrecha verterá menos agua que el que tenga la manguera ancha.

Si quisiéramos que saliera la misma cantidad de agua por las dos mangueras, podríamos poner las dos mangueras de igual anchura o bien aumentar la cantidad de agua del tanque (su carga) que tenga la manguera más fina, de modo que aumentará su presión y con ello su flujo de agua a través de la manguera.

Del mismo modo, si aumentamos el voltaje en un circuito, aumentará su corriente.

Ya tenemos la relación entre el voltaje (presión) y corriente (flujo de agua) pero hemos visto que influye un tercer elemento que es la anchura de la manguera; esta anchura representaría la resistencia en los circuitos eléctricos.

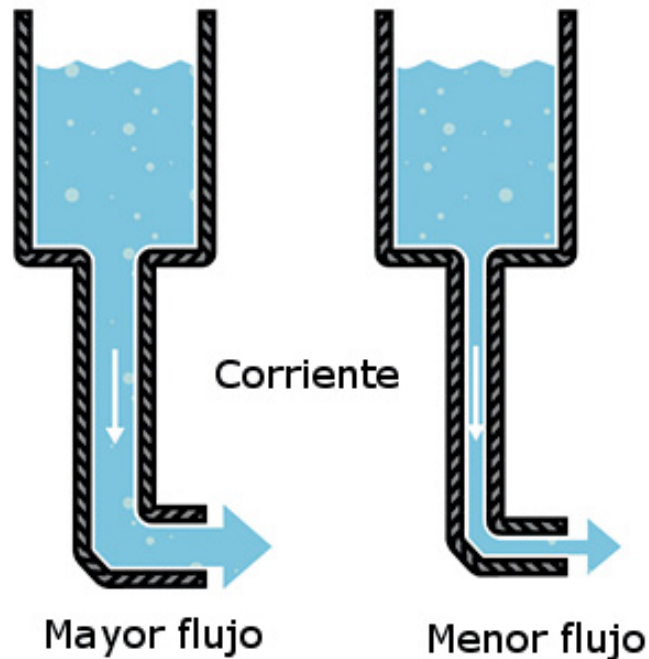


Figura 3.4. Equivalencia de la corriente con el agua

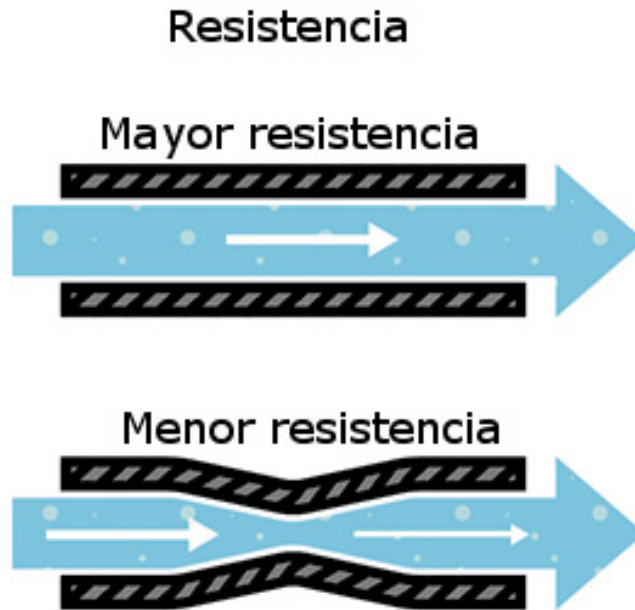
## Resistencia

Si volvemos al ejemplo anterior en el que cada uno de los tanques tenía una manguera más ancha que el otro, está claro que podemos expulsar más agua con la manguera más ancha que con la estrecha si la presión es la misma, esto nos muestra la resistencia. La manguera más estrecha se resiste a que el agua fluya por ella incluso teniendo en su entrada la misma presión que la manguera ancha.

En electricidad, sería tener dos circuitos con el mismo voltaje pero distintas resistencias; el circuito con mayor resistencia permite menor flujo de carga, por lo que tiene menor corriente, menos amperios.

La ley de George Ohm define la unidad de resistencia como la resistencia entre dos puntos de un conductor donde aplicando 1 voltio genera un flujo de 1 amperio y se representa por la letra griega omega  $\Omega$ .





**Figura 3.5.** Equivalencia de la resistencia con el agua

## **Relación voltaje, corriente y resistencia**

Cuando pensamos en electricidad y en electrónica, tenemos que tener en cuenta que los conceptos vistos anteriormente están unidos entre sí. Recordemos estos conceptos y sus equivalentes en el ejemplo del agua:

- Voltaje, diferencia de potencial o tensión (medida en voltios): Presión ejercida por el agua.
- Corriente o intensidad (medida en amperios): Flujo de agua.
- Resistencia (medida en ohmios): Grosor de la manguera.

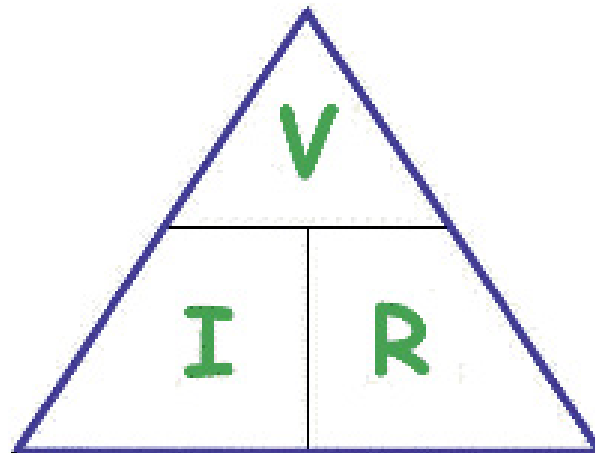
Pero... ¿de qué manera se relacionan? Pues mediante una de las leyes más importantes de la electricidad: la ley de Ohm. Esta ley nos indica la relación que existe entre la corriente el voltaje y la resistencia del modo:

$$I = V/R$$

Que dicho de otro modo sería:

$$V = IR \text{ o } R = V/I$$

Una manera sencilla de acordarse de estas fórmulas es tener en cuenta el triángulo de la figura 3.6



**Figura 3.6.** Triángulo de relación de Ohm

Con esto tenemos que si se disminuye la resistencia de un circuito, para que se sigan cumpliendo las igualdades, la corriente aumenta, o por ejemplo si se dobla la tensión en una resistencia, la corriente también se dobla.

Para la medición de estos parámetros existen unos aparatos específicos para cada uno de ellos, pero más práctico es uno llamado polímetro o multímetro, que permite medirlos todos e incluso ofrece otras funcionalidades.

Antes de continuar, conviene aclarar un concepto sobre la electricidad que es la corriente continua y corriente alterna. Dicho de un modo rápido, en la corriente alterna los electrones entre dos puntos A y B viajan en ocasiones de A a B y en otras ocasiones de B a A, intercalándose la polaridad positiva y negativa en dichos puntos, en cambio en la continua, siempre viajan o de A a B o de B a A. Un ejemplo de corriente alterna son los enchufes de casa que tienen 220 voltios y como ejemplo de corriente continua tenemos las pilas del mando de la televisión, que normalmente son de 1,5 voltios. Nosotros trabajaremos con corriente continua.

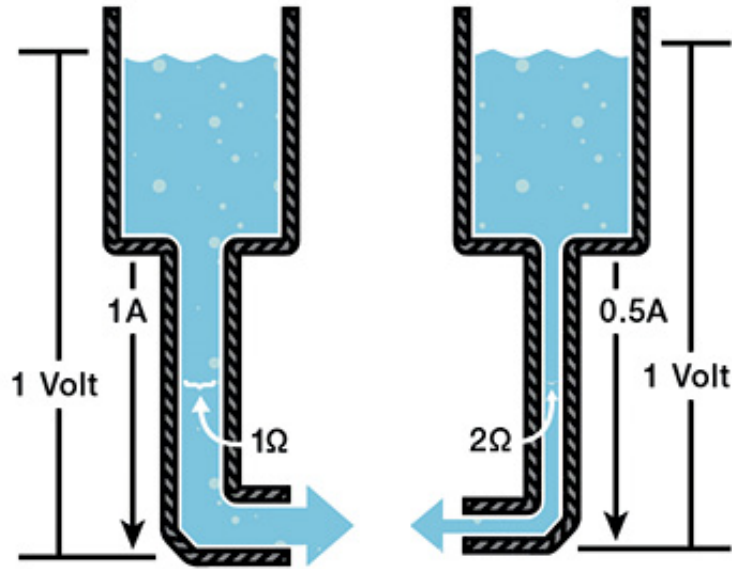


Figura 3.7. Tanques de agua y circuitos

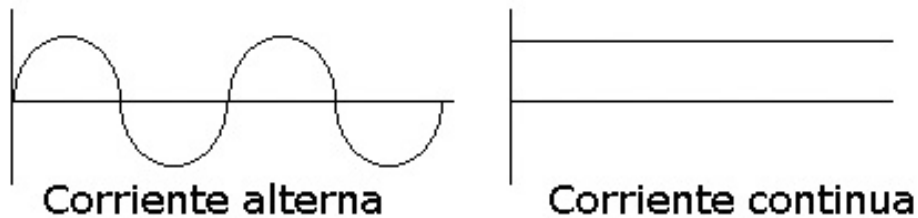


Figura 3.8. Diferencia entre corriente alterna y continua

## Abstracción del circuito

Si volvemos al montaje del circuito visto anteriormente donde tenemos una pila, conectada a una bombilla y un interruptor que hace que se encienda la bombilla; si lo quisiéramos representar sobre un papel, podríamos dibujar con todo detalle la pila, la bombilla y el pulsador, pero... ¿qué tipo de bombilla? ¿La hacemos redonda o con forma alargada? ¿Y el pulsador? ¿Cómo es el botón? Lo que se hace, en lugar de dibujar fielmente el montaje, se dibuja una abstracción, y así cada elemento puesto en el montaje tiene su correspondiente dibujo fácil de recordar y sencillo de dibujar. A estos dibujos se les rodea de información útil a la

hora de realizar el montaje final, por ejemplo en el caso de la pila, es necesario poner el voltaje que debe tener.

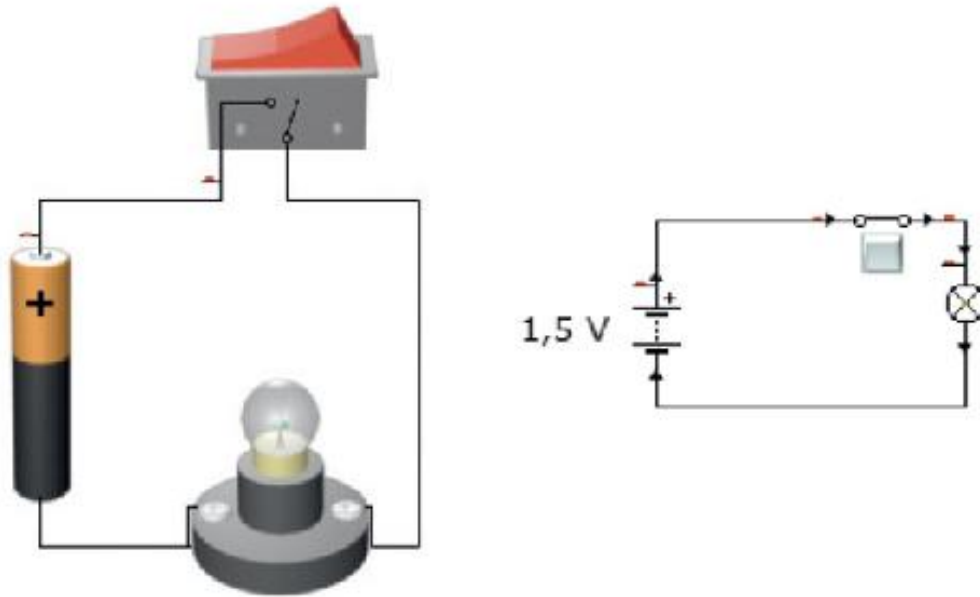


Figura 3.9. Ejemplo de circuito

Durante el libro veremos unos cuantos circuitos donde se irán mostrando también los símbolos de los componentes que vayamos utilizando.

## ¿Qué es la electrónica?

Es la parte de la ciencia que estudia los electrones y su comportamiento, pero también es el campo de la ingeniería y de la física que se encarga del diseño y aplicación de dispositivos y circuitos electrónicos, cuyo funcionamiento depende del flujo de electrones para la generación, transmisión, recepción, almacenamiento de información, entre otros.

Como podemos ver, la electrónica está estrechamente ligada a la electricidad, ya que sin electricidad no hay electrones que se muevan y por lo tanto no funcionarían los componentes electrónicos.

Dentro de la electrónica podemos distinguir entre la electrónica analógica y la digital.

- **Electrónica analógica:** es la que trabaja con valores continuos, por ejemplo temperaturas, distancias, etc. Podemos tener 4.3 metros o 4.345 metros. Al pasar de una unidad a otra no hay continuidad, como si fuera una rampa.
- **Electrónica digital:** es la que trabaja con valores discretos, por ejemplo número de libros, o personas. Podemos tener 4 personas o 5, pero no 4.345 personas.

Trabajando con electrónica analógica nos encontraremos que es muy difícil de guardar sus datos, manipularlos o simplemente compararlos mientras que con electrónica digital todas estas tareas son muy sencillas y además, casi todos los circuitos de electrónica analógica pueden ser fácilmente transformados en digital. Es por eso que la electrónica digital se va utilizando cada día más.

En nuestros circuitos con Arduino, los valores de resistencias, voltajes y corrientes serán siempre valores continuos, pero en ocasiones necesitaremos trabajar con ellos de manera discreta, de manera digital.

## **Electrónica digital**

Cuando trabajamos con electrónica digital, podemos tener tantos valores discretos como queramos, pero lo normal es que se trabaje solo con dos valores, los conocidos como 0 y 1, falso y verdadero (*false* y *true*), bajo y alto (*low* y *high*).

Hemos dicho que trabajamos con electricidad, pero el voltaje es un valor continuo y podemos tener por ejemplo 3V, 3.77V ¿cómo lo pasamos a discreto? Pues se establece un valor de referencia de modo que si el voltaje es menor que ese valor, entonces es 0 y si es mayor es 1.

Nosotros trabajaremos normalmente entre 0 y 5V, de modo que valores de 0V y cercanos a 0V representarán el valor bajo (*low*) y cercanos a 5V será

el valor alto (*high*). Al no tener que ser 0 y 5 exactamente, permite que los circuitos puedan tener pérdidas o no tener componentes muy precisos.

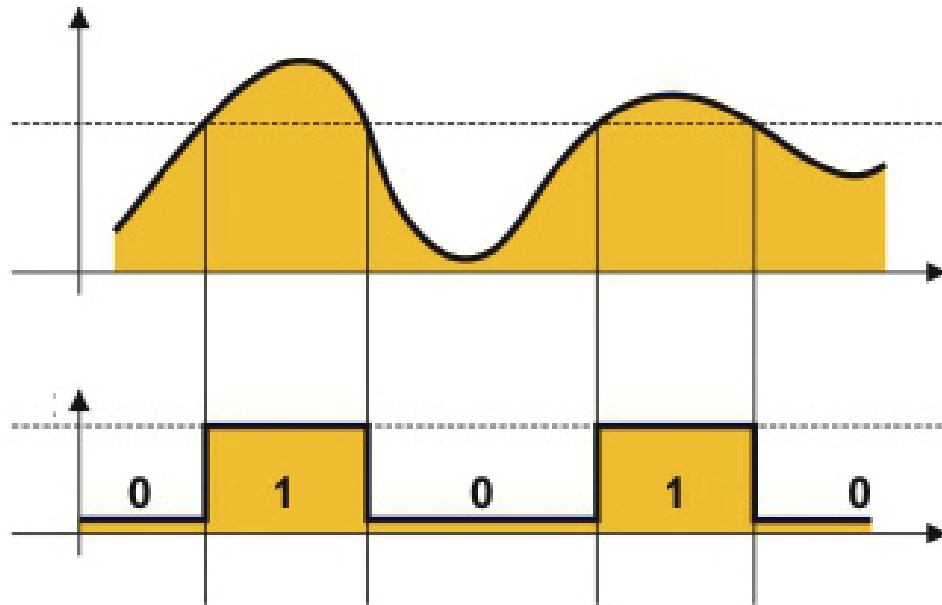


Figura 3.10. Paso de valor continuo a discreto

## Binario

Cuando trabajamos con electrónica digital, trabajamos con valores 0 y 1, esto quiere decir que cualquier dato o valor, debemos ser capaces de representarlo mediante estos dos números. Para conseguir esta tarea se utilizan números en base 2, lo que se conoce como sistema binario.

En electrónica normalmente se asigna el 1 al estado de tensión más alto y 0 al más bajo, pero pueden encontrarse sistemas de lógica inversa o lógica negativa cuyo estado de tensión más alto corresponde al 0 y el más bajo al 1.

Cada uno de los dígitos binarios se denomina *bit*, a los grupos de 8 bits se les denomina *bytes*, que es un término comúnmente utilizado; otra agrupación no tan utilizada es la de 4 bits llamada *nibble*.

Cada cantidad decimal se puede representar mediante una cantidad binaria y viceversa, mediante la transformación entre bases se obtienen las

cantidades representativas en cada una de las bases seleccionadas.

No vamos a entrar mucho en el sistema binario, pero saber que el mayor número de cantidades que puede representar en binario un grupo de bits viene dado por el número de bits que haya, concretamente para n bits el mayor número de cantidades representable es  $2^n$ , es decir si tenemos 2 bits podremos representar hasta  $2^2 = 4$  cantidades, en este caso 0, 1, 2 y 3; dicho de otro modo, podemos representar hasta el  $2^n-1$ ; el número de posibles cantidades a representar se dobla por cada nuevo bit que añadamos. Así por ejemplo el número 135 en binario es 10000111.

1 bit	2 bits	3 bits	4 bits
0	00	000	0000
1	01	001	0001
	10	010	0010
	11	011	0011
		100	0100
		101	0101
		110	0110
		111	0111
			1000
			1001
			...
			1111
$2^1=2$	$2^2=4$	$2^3=8$	$2^4=16$

---

**NOTA:** El número de combinaciones y por tanto de elementos a representar con n bits son  $2^n$ .

---

Las letras y otros caracteres que podamos mostrar por pantalla o introducir en un teclado, también son guardados en memoria utilizando base 2, esto se hace mediante una tabla de conversión llamada ASCII, en la que se hace corresponder cada carácter con un valor numérico y luego ese valor numérico pues se puede guardar ya en base 2; por ejemplo el carácter “@” se almacena mediante el número 64 (40 en hexadecimal) y el carácter “3” se almacena mediante el número 51 (33 en hexadecimal), lo que hace que si en memoria tenemos el número 64, puede realmente ser el número 64 y poder hacer con él sumas y restas, o ser la representación de “@” con lo que nos serviría para escribir un correo. La tabla ASCII está fácilmente disponible en internet.

Otras bases muy utilizadas en electrónica son la base 8 y la base 16 (por ser 1 y 2 bytes concretamente). El problema que se tiene con la base 16 es que nos faltan números para poder representar el valor, así que lo que se hace es utilizar letras.

<b>Decimal</b>	<b>Hexadecimal</b>	<b>Binario</b>
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001



10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Con las restricciones de la base 2 y el hecho de no poder guardar en un bit de memoria si es positivo o negativo un número (recordemos que solo podemos guardar 0 y 1) nos lleva a distintas representaciones binarias, lo que hace que un mismo número, dependiendo de la representación o notación usada, sea un valor u otro; así por ejemplo tenemos la notación complemento a 1, complemento a 2, BCD, etc...

Es sencillo hacer transformaciones entre bases, pero lo más útil es utilizar alguna calculadora que tenga esta opción, como la propia de Windows.

## Operaciones a nivel bit

Cuando trabajamos con los valores de los bits directamente, es decir solo con los valores 0 y 1 de manera unitaria. A estas operaciones también se les denomina operaciones lógicas. En las operaciones a nivel de bit tenemos:

- **NOT:** Es la operación “no”. Solo utiliza un operando, es decir un solo bit y lo que hace es cambiar su valor, de modo que si el bit tiene valor 1, pasa a ser valor 0 y viceversa
- **AND:** Es la operación “y”. Necesita dos operandos y para que el resultado sea 1, necesita que los dos bits sean 1.

**Bit1**

**Bit2**

**Bit1 AND Bit2**

0	0	0
0	1	0
1	0	0
1	1	1

- **OR:** Es la operación “o”. Necesita dos bits y el resultado será 1 siempre y cuando alguno de los bits operandos, tenga el valor 1.

Bit1	Bit2	Bit1 OR Bit2
0	0	0
0	1	1
1	0	1
1	1	1

- **XOR:** Es la operación “o exclusiva”. Necesita dos bits y el resultado será 1 siempre y cuando solo uno de los bits operandos, tenga el valor 1.

Bit1	Bit2	Bit1 XOR Bit2
0	0	0
0	1	1
1	0	1
1	1	0

Podemos encontrarnos también con otras operaciones que son composición de las anteriores, como por ejemplo NAND que sería aplicar un AND y un NOT.

Estas operaciones podemos usarlas con valores booleanos, es decir usando *true* y *false* en lugar de 1 y 0.

Aunque se aplican bit a bit, no quiere decir que no podamos aplicar estas operaciones a un byte (recordemos que son 8 bits) simplemente tenemos que ir aplicando bit a bit la operación.

10001110 XOR 11100011 = 01101101

# 4

## Programación



**En este capítulo aprenderá a:**

- Cargar un programa en la tarjeta
- Conocer la estructura de un programa
- Poner las bases de la programación Arduino
- Diferenciar los distintos bloques de control

Es hora de comenzar a hacer nuestros propios programas y ver qué posibilidades nos da Arduino para indicarle las órdenes a procesar. Lo que haremos en este capítulo es ver las bases de la programación que usaremos en los capítulos posteriores. Dado que la programación puede resultar difícil o pesada, en este capítulo (que quizá sea el más aburrido) simplemente veremos unas pinceladas, y mediante los ejemplos iremos ampliando el conocimiento de las instrucciones.

Para poder dar instrucciones a la placa Arduino, tenemos que hablarle en su idioma, que no es inglés, ni español ni nada de eso; para podernos comunicar con la placa Arduino necesitamos darle las instrucciones en *Arduino Programming Language*, que como veremos, con muy pocas instrucciones podemos crear programas muy interesantes.

## General

Los programas que se cargan en la tarjeta Arduino, son denominados *sketches* y están compuestos por una serie de líneas que son las instrucciones; en realidad se trata de unos ficheros de texto con las operaciones a ejecutar desde la tarjeta Arduino. Estos ficheros se componen a su vez de unos bloques de código que cada uno tiene una misión. Con tal de poder ver el aspecto de un programa, comenzaremos abriendo un programa ya hecho. Para ello pulsamos en el menú Archivo>Ejemplos>01.Basicos>Blink, esto abrirá un ejemplo de *sketch* que consiste en una serie de instrucciones que permiten que el led (una bombillita) de la tarjeta se ilumine de manera intermitente. El *sketch* tiene el aspecto:

```
/*  
...  
*/  
// the setup function runs once when you press reset or power the board  
void setup() {  
  // initialize digital pin 13 as an output.
```

```
pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
delay(1000); // wait for a second
digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
delay(1000); // wait for a second
}
```

Lo primero que vemos en este código es un texto entre los caracteres `/*` y `*/`, todo lo que esté entre estos caracteres se considera comentarios y la tarjeta no lo tendrá en cuenta. Los comentarios son muy importantes cuando realizamos programas, ya que como tenemos que escribir líneas que entienda Arduino, no son instrucciones que entendamos nosotros rápidamente y al cabo de unos días nos habremos olvidado de qué hacían esas líneas; por eso es importante escribir en lenguaje humano lo que se va haciendo en los programas.

Además de los caracteres `/*` y `*/` que permiten realizar bloques de comentarios, tenemos los caracteres `//` que permiten realizar una línea de comentario, es decir, desde que aparecen estos dos caracteres hasta el final de la línea, no es tenido en cuenta por la tarjeta Arduino. En el entorno de programación veremos que los comentarios se diferencian del código, porque se muestran en letra gris en lugar de negra.

En el ejemplo podemos ver dos bloques de código marcados entre los caracteres `{}`. Realmente estos bloques son funciones, que tienen como nombre `setup` y `loop` respectivamente. También aparece la palabra `void` delante de estos nombres de función; pero por ahora no la explicaremos.

El bloque `setup()` es en realidad una función que se ejecuta cada vez que se enciende la placa o se realiza un *reset* de la misma (pulsando el botón que vimos anteriormente en la placa). Se ejecuta una y solo una vez por cada encendido de la placa y nos sirve para configurar el uso de los pines y preparar el programa...

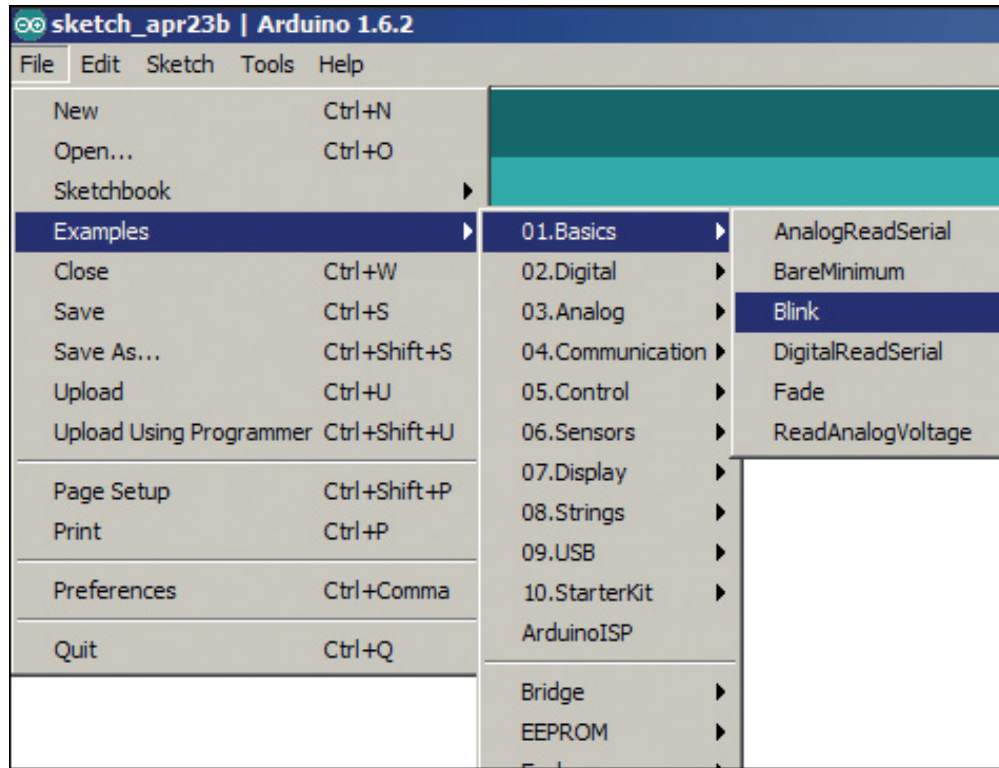
El bloque `loop()` es el corazón del *sketch*, como su propio nombre indica, ejecuta una y otra vez todas las instrucciones que se encuentren en su

interior en el orden en el que aparecen. Resumiendo, la tarjeta ejecutará el bloque `setup()` y tras él, ejecutará todo el tiempo el bloque `loop()` que es el que tiene el control de la placa Arduino y será dentro de él donde tendremos que leer las entradas de la tarjeta, procesar los datos y mostrar los datos o actuar sobre algún elemento externo.

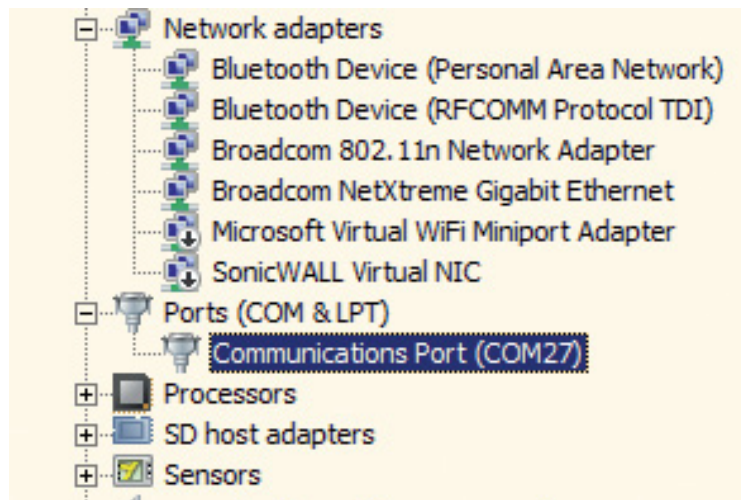
Realmente solo necesitamos los bloques `loop()` y `setup()` para que funcione (compile) el *sketch*, si bien casi con toda seguridad necesitaremos hacer uso de alguno bloque más con código propio.

Antes de continuar, vamos a probar el *sketch* en la tarjeta, de modo que podamos ver realmente lo que hace (figura 4.1).


Para poder cargar el *sketch* en la tarjeta, lo primero es conectar ésta al USB del ordenador mediante el cable correspondiente; al realizar esta acción se debe encender en la tarjeta una luz verde indicando que está recibiendo electricidad. Antes de poder ejecutar el *sketch* hay que asegurarse que se tiene seleccionada correctamente la tarjeta a utilizar mediante el menú Herramientas>Tarjeta. Del mismo modo se debe seleccionar el puerto al que está conectada la tarjeta mediante el menú Herramientas>Puerto. Si no se sabe el puerto en el que está conectada la tarjeta, puede consultarse desde el administrador de dispositivos que ya se vio durante la instalación de los drivers de Arduino.



**Figura 4.1.** Selección del *sketch* de ejemplo



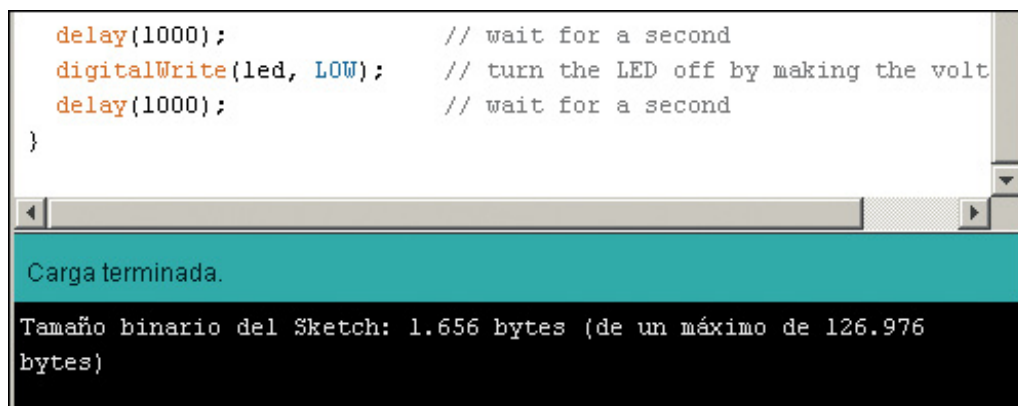
**Figura 4.2.** Tarjeta Arduino Uno conectada en el puerto serie COM27

Aunque en la figura 4.2 aparezca el puerto 27, no tiene porqué ser este, en cada caso será uno distinto. Pulsando el botón  o mediante el menú Archivo>Cargar, se procede al envío del programa a la tarjeta. Durante estos instantes aparecerán mensajes de información en el entorno de programación y el led correspondiente a la recepción en la placa (marcado



con la etiqueta Rx) se encenderá, dando a entender que está recibiendo datos. Una vez cargado el programa se podrá observar como un led de la tarjeta comienza a parpadear manteniéndose un segundo apagado y un segundo encendido de manera continuada. Si no fuera así, habría que revisar el error mostrado por la consola y ver si se ha seleccionado de modo correcto la tarjeta y el puerto de conexión.

Vamos a ver un poco más en los *sketches*.



```
delay(1000);           // wait for a second
digitalWrite(led, LOW); // turn the LED off by making the volt
delay(1000);           // wait for a second
}

Carga terminada.
Tamaño binario del Sketch: 1.656 bytes (de un máximo de 126.976 bytes)
```

**Figura 4.3.** Mensaje de carga completa en el entorno de desarrollo

## Formato

El formato genérico de todos los programas es:

```
//Añadir librerías si es necesario
//Listado de las variables que se necesiten

// función de configuración de la placa
void setup() {
    // Se configuran los pines necesarios
}

//función principal
void loop() {
    //se ejecuta periódicamente
}

//otras funciones
void function() {
    //Funciones personales
}
```

Como hemos visto, solo las funciones (bloques) `setup` y `loop` son necesarios para que pueda funcionar, el resto es optativo y lo iremos usando cuando vayamos avanzando.

Las librerías son unos programas que ya están hechos y que nos sirven para no reinventar la rueda. Por ejemplo si alguien ya ha programado la manera de enviar datos a una pantalla, mejor no volver a escribir nosotros las órdenes y usar las que sabemos que funcionan.

Tras las librerías, decimos las variables globales que vayamos a utilizar en el programa. Las variables nos servirán para mantener datos en memoria, por ejemplo si tenemos que encender una bombilla cuando se pulse tres veces un botón, usaríamos una variable que fuera guardando el número de veces que se ha pulsado el botón, y en el bloque `loop`, habría una instrucción que miraría si la variable tiene el valor igual a 3 y entonces encendería la bombilla. Las variables que indiquemos en esta zona, son las que se denominan globales, porque no están dentro de ningún bloque (bueno, aunque el fichero también lo podemos considerar un bloque) y son visibles desde cualquier parte de código del fichero (más adelante ya comprenderá lo que implica).

Por último el bloque `function()` nos sirve para indicar todas las funciones adicionales que queramos crear, aunque en un punto posterior se trata con más profundidad, una función es un bloque de código que se puede llamar en el programa desde distintos puntos.

Si nos fijamos en las instrucciones u órdenes de las cuales se compone el *sketch* que hemos abierto de ejemplo, podremos ver que cada una de ellas acaba con un carácter “;” (punto y coma). Este carácter indica el fin de una instrucción. En este caso, tenemos una instrucción por línea, pero podríamos haber escrito todo el programa en una línea, siempre y cuando separemos cada instrucción con “;”. Las instrucciones siguientes sirven para definir tres variables:

```
const int buttonPin = 2;
const int led1Pin = 4;
const int led2Pin = 5;
```

Pero se podrían haber escrito:

```
const int buttonPin = 2; const int led1Pin = 4; const int led2Pin = 5;
```

O incluso:

```
const int buttonPin = 2; const int led1Pin  
= 4  
; const int  
led2Pin = 5;
```

Aunque resulta mucho más difícil de leer.

El entorno de programación de Arduino proporciona una utilidad para ordenar el código y dejarlo de modo que lo podamos leer sin problemas, se realiza mediante el menú Herramientas >Formato Automático o mediante la combinación de teclas **Ctrl -T**.

## Las variables

Las variables son la parte del código que sirve para almacenar datos en memoria y trabajar más adelante con ellos. Antes de poder usar una variable, hay que decir cómo se va a llamar y qué tipo de dato va a guardar (no es lo mismo guardar un número que un texto) a esto se le llama declarar la variable.

Las variables se identifican por un nombre que puede ser desde una letra hasta una frase entera, pero siempre teniendo en cuenta que el nombre debe empezar por una letra o el carácter “\_”, siendo el resto del nombre números o letras, mayúscula o minúscula y sin espacios. Algunos ejemplos de nombres de variables pueden ser: `i`, `pin2`, `ledError`, `numero_de_vueltas...` Hay que tener en cuenta que Arduino es sensible a las mayúsculas, es decir para él, `ledError` y `lederror` no son la misma variable (es lo que se llama *case sensitive*). Es muy aconsejable utilizar nombres descriptivos para las variables, aunque en contadores suelen utilizarse variables de una sola letra, normalmente la `i`.

Cuando se declara una variable dentro de un bloque, esta no está disponible para todo el programa, no es una variable global que podamos leer y escribir desde cualquier punto del código sino que solo se puede ver en el lugar donde se declara. Por ejemplo las variables que se declaran fuera de cualquier función, sí son visibles desde cualquier punto y son llamadas variables globales o de programa, mientras que las que se declaran dentro de las funciones son solo se pueden usar desde dentro de la propia función. Para ser más precisos, las variables son solamente visibles dentro del bloque en las que se declaran, así si se declara una variable dentro de un bloque de tipo bucle, solo se puede acceder a ella desde el propio bucle y no desde fuera.

Además del nombre, tenemos que indicar el tipo de dato que va a guardar; si es un carácter o un número o cómo de grande es el número.... Esto sirve para dos cosas:

- Permite al compilador (el programa que se encarga de verificar el programa) tener una lista de variables con el tipo de cada una y poder comprobar en busca de errores, por ejemplo si intentamos multiplicar un texto por un número.
- Hacer que el compilador pueda tener información sobre la cantidad de memoria que va a necesitar para cada variable cuando se ejecute el programa y el tipo de operaciones y transformaciones que podemos hacer con ella.

Los tipos de variables se representan por una serie de palabras reservadas (que no podemos usar como nombre de variables ni funciones). Veremos ahora unos pocos y a lo largo del libro ya conoceremos alguno más:

- `void`: Es el tipo nulo. Se usa solo en la declaración de las funciones, significa que la función no devolverá nada cuando retorne después de su ejecución.
- `boolean`: Sirve para guardar verdadero o falso, solamente puede tener dos valores *true* o *false*. Ocupa un byte de memoria. Ej:  
`boolean error = false;`

- `char`: Contiene un carácter. En memoria ocupa un byte. Para informarlo se debe utilizar la comilla simple, guardando la comilla doble para las cadenas de texto, es decir 's' es un carácter, pero "s" es una cadena de texto y no se puede guardar en una variable de este tipo. Al guardar un carácter, lo que se está guardando realmente es un número que representa a ese carácter, lo cual permite realizar operaciones aritméticas (por ejemplo para pasar a mayúsculas). Ej: `char unChar = 'X'; char other = 64 // es la @`
- `byte`: Guarda un valor numérico de 0 a 255. En memoria ocupa (como su nombre indica) 1 byte (8 bits) se puede indicar su valor en diferentes bases numéricas. Ej: `byte years = B100001; //33 en decimal`
- `int`: En memoria ocupa 2 bytes (16 bits) podemos guardar valores entre -32,768 y 32,767. Ej: `int years = -33;`
- `unsigned int`: Es semejante al tipo `int` con la diferencia que guarda solamente números positivos. Ocupa 2 bytes en memoria y en este caso puede guardar valores numéricos desde 0 hasta 65,535. Ej: `unsigned int years = 33;`

## Operaciones

Una operación es una manipulación de una variable o varias con tal de modificar su contenido o compararlo con algo. Arduino nos proporciona todas las operaciones necesarias o herramientas para crearlas. Podríamos clasificar las operaciones en aritméticas, de comparación, con booleanos, a nivel de bit y de composición entre aritméticas o nivel de bit y de asignación.

Dentro de las operaciones aritméticas tenemos:

- `=`: Es el operador de asignación, da a la variable situada a la izquierda del operador el valor de lo situado a la derecha. No se debe confundir con el operador de comparación `==`. Ej: `int i =8; //da el valor 8 a la variable i.`

- +: Permite sumar dos operandos
- -: Permite restar dos operandos
- \*: Permite multiplicar dos operandos
- /: Permite dividir dos operandos
- %: Es el operador llamado módulo. Obtiene el resto de una división entera. Ej: `i = 9 % 4; // i tendría el valor de 1.`

---

**NOTA:** A la hora de ejecutar las operaciones Arduino utiliza los tipos de los operandos para obtener el resultado, así al realizar `5 / 2` se obtendría como resultado `2` como entero, por lo que se debe tener cuidado con los tipos seleccionados en cada operación. También se debe vigilar lo que se denomina “desbordamiento” por tipo de dato; el tipo `int` puede almacenar valores de `-32,768` a `32,767`, eso quiere decir que si tenemos una variable entera que tenga de valor `32,767` y le sumamos `1`... automáticamente pasa a tener de valor `-32,768` por desbordamiento.

---

Las operaciones de comparación son:

- `==`: Igualdad; compara si los operandos a cada lado de la operación son iguales o no y devuelve `true` en caso de ser iguales y `false` en caso de no serlo.
- `!=`: Desigualdad; compara si los operandos a cada lado de la operación son diferentes o no y devuelve `true` en caso de ser distintos y `false` en caso de ser iguales.
- `>`: Mayor que; compara si el operando situado a la izquierda operación es mayor que el de la derecha y devuelve `true` en caso de ser mayor y `false` en caso de no serlo.
- `>=`: Mayor o igual que; compara si el operando situado a la izquierda operación es mayor o igual que el de la derecha y devuelve `true` en caso de ser mayor o igual y `false` en caso de no serlo.
- `<`: Menor que; compara si el operando situado a la izquierda operación es menor que el de la derecha y devuelve `true` en caso de ser menor y `false` en caso de no serlo.
- `<=`: Menor o igual que; compara si el operando situado a la izquierda operación es menor o igual que el de la derecha y devuelve `true` en caso de ser menor o igual y `false` en caso de no serlo.

Las operaciones booleanas son aquellas que trabajan con variables de tipo booleano, solo aceptan *true* y *false* como valores. Las operaciones booleanas disponibles son:

- `&&`: Es el “y” booleano. Se obtiene *true* si los dos operandos son *true*.
- `||`: Es el “o” booleano. Se obtiene *true* si alguno de los dos operandos es *true*.
- `!`: Se trata del operador “no” booleano. Solo trabaja con un operando que se sitúa a su derecha. Se obtiene *true* si el operando es *false*.

Las operaciones a nivel de bit:

- `&`: Es el “y” lógico. Devuelve 1 cuando los dos bits que entran en juego son 1.
- `|`: Se trata del “o” lógico. Devuelve 1 cuando uno de los bits que entran en juego es 1.
- `~`: Es el “no” lógico. Devuelve 1 cuando el bit que entra en juego es 0.
- `^`: Es el “o exclusivo” lógico. Devuelve 1 cuando uno y solo uno de los bits que entran en juego es 1.
- `<<`: Realiza un desplazamiento de bits a la izquierda, introduciendo 0 por la derecha.
- `>>`: Realiza un desplazamiento de bits a la derecha. El valor que se introduce por la izquierda depende de si el tipo de dato es con signo o sin signo.

Antes de seguir, vamos a ver unos pequeños ejemplos sobre las operaciones a nivel de bit para que queden más claras:

```
int a = 13; // en binario 0000000000001101
int b = 6; // en binario 0000000000000110
int z = a & b; //z = 0000000000000100
int y = a | b; //y = 0000000000001111
int x = a ^ b; //x = 0000000000001011
int w = ~a ; //w = 1111111111110010 o -14
```

```
int v = a << 3; // v = 0000000001101000 o 104 que son 13 * 2^3, 3 son las posiciones
que hemos desplazado
int u = a >> 3; // v = 0000000000000001 o 1 que son 13 / 2^3, 3 son las posiciones que
hemos desplazado
```

Tanto en el caso de desplazamiento a la izquierda como a la derecha, los bits que “salen” se pierden.

Las operaciones compuestas son aquellas que con una operación podemos hacer dos, por ejemplo sumar y asignar el valor de la suma. Los operadores disponibles son:

- **++:** Incrementa en 1 la variable, `++myVar;` sería semejante a `myVar = myVar + 1;`. El operador puede ir antes o después de la variable, dependiendo de donde vaya actúa de modo diferente. Si precede a la variable, primero incrementa la variable y luego devuelve su valor, si va detrás primero devuelve el valor y luego incrementa la variable.

```
int myVar = 10;
int myInc = ++myVar; // myInc tiene valor 11 y myVar tiene valor 11
int myInc2 = myVar++; // myInc2 tiene valor 10 y myVar tiene valor 11
```

- **--:** Semejante a `++` pero reduciendo en 1 el valor de la variable.
- **+=:** Asignación con suma. Permite sumar y asignar a la vez. `myVar += 10;` es lo mismo que `myVar = myVar + 10;`
- **-=:** Asignación con resta. Permite restar y asignar a la vez. `myVar -= 10;` es idéntico a `myVar = myVar - 10;`
- **\*=:** Asignación con multiplicación. Permite multiplicar y asignar a la vez. `myVar *= 10;` es lo mismo que `myVar = myVar * 10;`
- **/=:** Asignación con división. Permite dividir y asignar a la vez. `myVar /= 10;` es igual que `myVar = myVar / 10;`
- **&=:** Asignación con “y” a nivel lógico. Realiza un “y” lógico y se lo asigna a la variable. `a &= b;` es lo mismo que `a = a & b;`
- **|=:** Asignación con “o” a nivel lógico. Realiza un “o” lógico y se lo asigna a la variable. `a |= b;` es igual que `a = a | b;`



# Bloques de control

Ya sabemos cómo se guardan los datos en memoria para trabajar con ellos e incluso conocemos la manera de realizar algunas operaciones pero nos falta saber cómo controlar qué acciones realizar en cada momento. Estos bloques de control del programa que nos servirán para indicar al procesador que realice una acción un número determinado de veces, controlar cuando ejecutar unas acciones u otras dependiendo de algo, etc.

En este apartado se nombrarán alguno de estos bloques pero para no aburrirnos, dejaremos la explicación de su funcionamiento para los ejemplos que veamos más adelante.

## **if**

El bloque `if` permite ejecutar una sección de código cuando se cumpla una condición dada, por ejemplo si el contador llega a 3, encender una luz. Su estructura es:

```
if (condición){
    //código a ejecutar
}
else{
    //código a ejecutar si no se cumple la condición
}
```

## **switch**

Los bloques *switch* permiten ejecutar diferentes códigos dependiendo de los distintos valores que pueda tomar una variable o expresión. Sería una especie de `if` anidado o muchos `if` seguidos. Supongamos que tenemos una variable con el mes del año y queremos sacar por pantalla el nombre del mes dependiendo del valor que tiene esa variable. Se podría usar un `if... else` con once comprobaciones más un `else` “por defecto”, pero sería muy largo de escribir; en lugar de esto podemos usar la orden

`switch`, donde se genera un bloque de código por cada valor que pueda tener la variable y se ejecutará el bloque correspondiente al valor que tenga la variable en ese momento. Sería:

```
switch (variable_a_comprobar){
  case valor1:
    //código valor 1
    break;
  case valor2:
    //código valor 2
    break;
  default:
    //código valor por defecto
}
```

## **goto**

Esta instrucción permite saltar a una parte de código previamente marcada con una etiqueta. Su uso es:

```
etiqueta1:
//código variado...
goto etiqueta1;
```

Con esto se haría que al llegar a la instrucción `goto etiqueta1;` la ejecución volviera a la línea `etiqueta1`.

---

**NOTA:** El uso de esta instrucción está altamente desaconsejado por hacer el código muy difícil de mantener y crear situaciones de error a medida que se introducen más sentencias de este tipo en el código.

---

## **Bucles**

Los bucles nos permitirán ejecutar una serie de instrucciones un número de veces. El número de veces que se ejecutan estos bucles depende de la naturaleza del bucle y de los datos; podemos hacer que un bucle se repita un número determinado de veces o que se repita mientras haya una condición que sea verdad.

Los bloques de tipo bucle de los que disponemos son:

- `for`: Sirve normalmente para efectuar un número de ciclos conocido, que puede venir dado por un número fijo o uno variable, por ejemplo el número de letras de una palabra.
- `while`: Ejecutará un bucle mientras la condición que se le dé sea válida, pero antes de ejecutar el bucle, mira a ver si la condición es válida y si lo es, lo ejecuta.
- `do...while`: Parecido al anterior, ejecutará un bucle mientras la condición que se le dé sea válida, pero primero ejecuta el bucle y luego mira a ver si la condición es válida y si lo es, vuelve a ejecutar el bucle.

Más adelante cuando veamos los ejemplos nos quedarán claros los funcionamientos de cada uno de los bloques.

## Funciones

Una función es un bloque de código que realiza una tarea particular. Trabajar con funciones es una buena manera de mantener el código limpio y ordenado, creando bloques con funcionalidad propia aislados del resto de código. Podemos decir que una función sirve para cumplir dos propósitos: hacer que el programador pueda decir “este bloque de código hace esto, solo esto y específicamente esto y no debe mezclarse con el resto de código” y hacer que el bloque pueda usarse en diferentes partes del código.

No vamos a entrar más en detalle que ya hemos visto mucha teoría. Es hora de ensuciarse las manos.

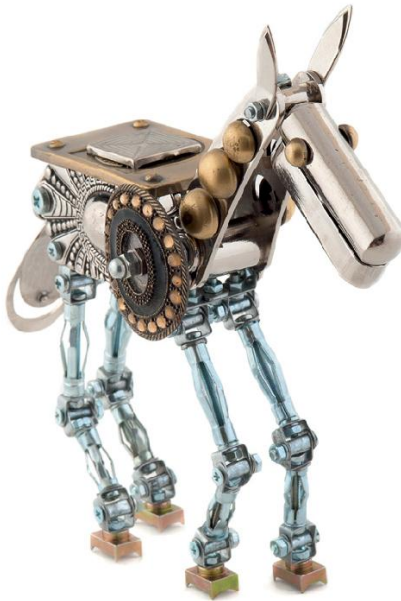
### Instrucciones específicas

Existen una serie de instrucciones que usaremos dentro de los bloques que hemos visto anteriormente. Estas instrucciones nos servirán por ejemplo para leer el valor que exista en alguna de las entradas de la tarjeta, realizar

una escritura en las salidas, detener el programa unos instantes... Las iremos viendo a lo largo del libro con cada uno de los ejemplos. Y ya ha llegado el momento de ponerse manos a la obra.

# 5

## Primeros programas



**En este capítulo aprenderá a:**

- Utilizar el monitor serie
- Crear un *sketch* desde cero
- Manejar las sentencias de control
- Crear un conversor de números
- Diferenciar los distintos tipos de bucles

Es hora de comenzar a realizar nuestros primeros *sketches*, En este capítulo pondremos a prueba lo visto en el capítulo anterior realizando unos cuantos programas que nos obligará a introducir nuevos comandos e instrucciones. Además, haremos uso del monitor serie disponible en el entorno Arduino emulando entradas y salidas de la tarjeta.

## Primer programa

Para estos primeros programas, vamos a dejar de lado los componentes electrónicos y vamos a jugar directamente con la tarjeta Arduino. Hemos hablado de un monitor serie que incorpora el entorno de desarrollo, vamos a usarlo para comunicarnos con la tarjeta.

Para poder trabajar con el monitor serie, lo primero que debemos hacer es configurar la conexión dentro del bloque correspondiente a la función `setup()`, para ello, usaremos la instrucción `Serial.begin(velocidad)`; donde `velocidad` es el número de bits por segundo que enviaremos. En cuanto al número a utilizar, para comunicarse con el PC podemos usar 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, o 115200 bps, aunque realmente podemos usar cualquier valor ya que esta conexión la podemos usar para comunicarnos con otros componentes. Del mismo modo, para desconectar la comunicación serie tenemos que usar la instrucción `Serial.end()`.


```
void setup() {  
    // se configura el puerto serie para trabajar a 9600 bps  
    Serial.begin(9600);  
}
```

Como ya tenemos configurada la conexión, podemos usarla para enviar datos, por ejemplo para escribir en el monitor, podemos usar `Serial.print()` o `Serial.println()`, la diferencia entre ellas es que la segunda termina la línea y lo siguiente que escribiéramos aparecería en la

línea de abajo. A estas funciones se les debe indicar lo que se quiere imprimir y opcionalmente el formato en el que quiere mostrarse. Vamos a crear un nuevo proyecto. Para ello vamos al menú Archivo > Nuevo y escribimos el código:

```
void setup() {  
  // se configura el puerto serie para trabajar a 9600 bps  
  Serial.begin(9600);  
}  
void loop() {  
  int year = 2010;  
  Serial.print("2010 en hexadecimal se escribe:");  
  Serial.println(year, HEX);  
  delay(500); //se detiene el programa medio segundo antes de un nuevo ciclo.  
}
```

Hemos usado también la función `delay()` que permite detener la ejecución del programa durante los milisegundos que le indiquemos, en este caso 500 milisegundos, medio segundo.

Antes de poder cargar el *sketch* en la tarjeta, debemos guardar nuestro trabajo; si no lo hemos hecho antes, al intentar cargar el programa nos saldrá una ventana indicándonos que tenemos que salvar primero el *sketch*. Conectamos la tarjeta al PC y teniendo el modelo y puerto configurados, pulsamos el botón para cargar el *sketch* en la tarjeta tal y como hemos hecho anteriormente. Una vez cargado el programa en la tarjeta, si pulsamos sobre el botón del monitor serie  o sobre el menú Herramientas>Monitor serial, se abrirá la ventana de monitorización donde veremos aparecer la frase “2010 en hexadecimal se escribe: 7DA”;

## Un poco más de programación

Ahora que sabemos utilizar el monitor serie, vamos a aprovecharlo para conocer un poco más de alguno de los bloques que vimos en el capítulo anterior y comprender mejor su uso mediante ejemplos

## Control mediante if

Ya vimos que la estructura básica de `if` era:

```
if (condición){
    //código a ejecutar
}else{
    //código a ejecutar si no se da la condición
}
```

La condición dada entre los paréntesis puede ser en sí una variable o una comprobación del tipo `a > b`, `b != 0` o similar. Si la condición es distinto de 0 o se tienen valores *HIGH* o *true* en caso de booleanos, entonces se ejecutará el código entre que se encuentre entre las llaves que delimitan el bloque `if`. Si el código a ejecutar es dentro del bloque `if` es solamente una línea, entonces podemos no escribir las llaves pero es muy aconsejable usarlas siempre para evitar errores: ya que si el día de mañana se añaden nuevas líneas al bloque, si nos olvidamos de poner las llaves, solo se ejecutaría la primera línea y no tendría el comportamiento esperado. Si se decide trabajar sin llaves, lo que podemos hacer es escribir la orden en una sola línea para que quede claro que el bloque se compone solo de esa instrucción:

```
if (condición) sentencia_a_ejecutar;
```

en lugar de

```
if (condición)
    sentencia_a_ejecutar;
```

---

**NOTA:** Hay que tener cuidado de no caer en la trampa de hacer la comparación de igualdad usando el operador “=” que es el de asignación en lugar de “==” que es el de comparación. Dado que el bloque *if* se ejecuta siempre que la comprobación sea distinto de 0, la asignación dará siempre como verdadero a no ser que se esté asignando el valor 0; la sentencia `if (i=5) {...}` siempre se ejecutará.

---

Una variación del bloque `if` es la llamada `if... else` que se compone de dos bloques, el primer bloque `if` ya se ha visto y el bloque `else` que se



ejecutaría siempre que no se ejecute el bloque `if`; la ejecución del código solo entra en uno de los dos bloques y siempre entra en uno de ellos. El bloque `else` funciona del mismo modo que el `if`, ejecutando lo que encuentra entre llaves o solo la primera instrucción en caso de que no haya llaves. Por ejemplo

```
if (a > b){
  //código a ejecutar
}
else{
  //código a ejecutar cuando b >= a
}
```

Si fuera necesario, el bloque `else` puede tener nuevas comprobaciones permitiendo así tener una batería de comprobaciones

```
if (a > b){
  //código a ejecutar
}
else{
  if (a > b){
    //código a ejecutar cuando b > a
  }
  else{
    //código a ejecutar cuando b = a
  }
}
```

O escrito de modo más compacto:

```
if (a > b){
  //código a ejecutar
}
else if (a > b){
  //código a ejecutar cuando b > a
}
else{
  //código a ejecutar cuando b = a
}
```

Para probar lo visto vamos a realizar un nuevo programa con una serie de `if` para ver la salida que tendríamos.

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

```

void loop() {
  int i = 10;
  if (i < 5){
Serial.println("Es menor de 5");
  }
  else{
Serial.println("Es igual o mayor de 5");
  }
  if (i == 10)Serial.println ("Es igual a 10");
  delay(1000);
}

```

En la salida veremos que se escribe cada medio segundo:

```

Es igual o mayor de 5
Es igual a 10

```

## Control mediante switch

El `switch` actúa como si fuera una serie de `if` consecutivos; su aspecto es:

```

switch (variable_a_comprobar){
  case valor1:
    //código valor 1
    break;
  case valor2:
    //código valor 2
    break;
  default:
    //código valor por defecto
}

```

Cuando se llega a un *switch*, el programa evalúa el valor de la variable *variable\_a\_comprobar* y lo compara con los valores indicados por las etiquetas *case*. En caso de que el valor de la variable a comprobar y el valor de la etiqueta coincida, entonces comienza a ejecutar el código hasta encontrar una sentencia *break*; o el fin del bloque `switch`. La palabra reservada `break`; la encontraremos en varios bloques y su función es salir en ese preciso instante del bloque en el que se encuentre, dejando las variables tal y como están en ese momento. Es muy importante tener en cuenta que el hecho de que hasta que no encuentre una orden `break` o se acabe el bloque `switch`, se seguirá ejecutando código. Hagamos el sketch:

```

void setup() {Serial.begin(9600);}
void loop() {
int a = 2;
switch (a){
    case 1:
        Serial.println("codigo1");//código1 si a igual a 1
        break;
    case 2:
    case 3:
        Serial.println("codigo2");//código2 si a igual a 2 o 3
    case 5:
        Serial.println("codigo3");// código3 si a igual a 5
        break;
    default:
        Serial.println("codigo4");//código4 para el resto de valores de a
    }
    delay(1000); //esperamos un segundo antes de otro ciclo
}

```

En la salida del monitor serie veremos que se escribe cada segundo:

```

codigo2
codigo3

```

En este caso, si la variable `a` toma el valor 10, se ejecutaría el código `codigo4`, correspondiente a la etiqueta `default`; como `a` tiene el valor 2, ejecutará el código2 y código3 ya que al no haber un `break`; seguiría ejecutando el código marcado por la etiqueta del caso 5 hasta llegar al `break` es decir en el terminal serie veremos las salidas “codigo2” y “código3”. En este caso, la ejecución sería idéntica si el valor de `a` fuera 2 o 3.

## Control mediante for

La estructura del bucle `for` es:

```

for (inicialización; condición; instrucción) {
//código a ejecutar
}

```

La parte de `inicialización` sirve para inicializar variables que se utilicen dentro del bucle, esta parte se ejecuta una sola vez al comenzar el

bucle. La `condición` sirve para saber cuándo se debe abandonar el bucle y darlo por terminado; el bucle se seguirá ejecutando hasta que la condición sea 0 o `false`; la condición se comprueba cada vez que se va a ejecutar un nuevo ciclo del bucle. La parte de `instrucción` se ejecuta cada vez que se termina un ciclo y se suele utilizar para incrementar o disminuir variables, pero se puede poner en ella lo que se quiera. Resumiendo, su ejecución sería: se ejecuta la `inicialización`, cuando empieza el bucle por primera vez, se comprueba la `condición`, si es verdadera entonces se ejecuta el código del bloque `for`, cuando se termina la ejecución del bloque se pasa a ejecutar el código correspondiente a `instrucción` y se comienza un nuevo ciclo de bucle con la comprobación de la `condición` y así sucesivamente hasta que la `condición` sea falsa.

```
for (int j = 0; j < 10; j++){  
  //código a ejecutar 10 veces  
}
```

Para abandonar el ciclo actual y comenzar un nuevo ciclo podemos utilizar la instrucción `continue`, que lo que hace es finalizar el ciclo actual en el momento en el que se ejecuta esta instrucción y continuar con un nuevo ciclo. No hay que confundirlo con `break` que saldría del bucle por completo, sin acabar el resto de ciclos que le quedaran por realizar. Por ejemplo para ejecutar un código solo los ciclos impares podemos hacer que se ejecute la instrucción `continue` en los ciclos pares:

```
void setup() {Serial.begin(9600);}  
void loop() {  
  for (int i = 0; i < 10; i++){  
    if (i%2==0){  
      continue;  
    }  
    // código a ejecutar solo los ciclos impares  
    Serial.print("Soy impar: ");  
    Serial.println(i);  
  }  
  delay(1000);  
}
```

Aquí en el bucle hemos realizado una comprobación con un `if` para saber si la variable `i` en el ciclo en el que se está ejecutando es par o impar. Se obtiene si es par mediante el operador `%` que nos devuelve el resto de una división; si el resto de la división entre 2 es igual a 0 quiere decir que es par, y no mostraremos el mensaje, así que ejecuta la instrucción `continue` para comenzar el siguiente ciclo del bucle y que se incremente `i` en uno.

En la salida del monitor serie veremos que se escribe cada segundo:

```
Soy impar: 1
Soy impar: 3
Soy impar: 5
Soy impar: 7
Soy impar: 9
```

## Control mediante while

Su formato es:

```
while(expresión){
    // código a ejecutar
}
```

Al igual que pasa con el bloque `for` se puede cancelar un ciclo y continuar con el siguiente mediante la instrucción `continue` o acabar con el bucle mediante `break`. Hay que vigilar que dentro del bloque correspondiente al código del bucle haya algo que varíe la expresión a tener en cuenta para finalizar el bucle, ya que si no estaríamos ante un bucle infinito. Por ejemplo, este bucle se repetiría 10 veces:

```
void setup() {Serial.begin(9600);}
void loop() {
  int i = 10;
  while (i >0){
    // código a ejecutar 10 veces
    Serial.print("Ciclos restantes:");
    Serial.println(i);
    i--;
  }
  delay(1000);
}
```

En la salida del monitor serie veremos que se escribe cada segundo:

Ciclos restantes:10

Ciclos restantes:9

Ciclos restantes:8

Ciclos restantes:7

Ciclos restantes:6

Ciclos restantes:5

Ciclos restantes:4

Ciclos restantes:3

Ciclos restantes:2

Ciclos restantes:1

## Control mediante `do...while`

Es otro tipo de bucle muy semejante al bucle *while* pero en este caso, la condición se comprueba al final del bucle en lugar de al principio, es decir, se ejecuta al menos una vez. Cada vez que se acaba el bucle mira si tiene que empezar uno nuevo. Su formato:

```
do{  
    // código a ejecutar  
} while (condición);
```

Hay que recordar que el hecho de que se compruebe la condición al final de cada bucle hace que los bloque `do...while` **al menos se ejecutan siempre una vez**. Si tenemos el código:

```
void setup() {Serial.begin(9600);}  
void loop() {  
    int i = 10;  
    do{  
        // se ejecutará una vez  
        Serial.println("Estoy en el bucle");  
    } while (i < 5);  
}
```

```
}
```

Veremos que imprimirá el mensaje en el monitor serie, ya que se ejecuta el ciclo una vez antes de hacer la comprobación.

## Conversor de base

Vamos a utilizar lo aprendido para mejorar un poco nuestro conversor a hexadecimal, de manera que podamos decirle el número que queremos convertir y la base en la que queremos convertirlo. Para mantener el número a convertir y la base, usaremos dos variables, que las declararemos fuera de los bloques; una de ellos será de tipo entero y la otra de tipo *byte*.

```
unsigned int numberToConvert =0; // número a convertir
byte base = '\0'; //base a convertir
void setup() {
    // se configura el puerto serie para trabajar a 9600 bps
    Serial.begin(9600);
}
```

Para el código del bucle principal usaremos algo que hemos nombrado anteriormente pero que no hemos visto: las funciones. El bucle quedaría:

```
void loop() {
    //obtener el número a transformar
    getNumber();
    //obtener la base y convertir
    getBaseAndConvert();
    // se inicializan de nuevo las variables para un nuevo ciclo
    numberToConvert = 0;
    base = '\0';
}
```

Como vemos, usaremos una función para obtener el número y otra para obtener la base y convertir. Podríamos haber usado más funciones o menos, esto es cuestión de qué se elija la hora del diseño del programa; en cada caso serán mejores unas opciones y en otros, otras opciones distintas. Para escribir las funciones, podemos hacerlo tras el bloque `loop()`.

La función `getNumber()` leerá la entrada serie y la guardará en una memoria interna. Para saber la cantidad de datos que tiene esta memoria podemos utilizar la instrucción `Serial.available()`, de modo que si es más que 0, podemos saber que tiene datos. Lo que se guarda en la memoria, no es directamente el número pulsado, sino la representación ASCII del carácter que se selecciona (por ejemplo el carácter “1” en memoria viene representado por el código ASCII 49), por lo que hay que hacer una conversión de esta representación al número representado; para ello usaremos la función `Serial.parseInt()`. En esta función se espera que el usuario introduzca el número a transformar, por lo que haremos un bucle que se repetirá hasta que existan datos para leer en la memoria. La función quedaría:

```
/*
*****
* Obtiene el número a convertir
*****
*/
void getNumber(){
  Serial.println("Introduce el número a convertir:");
  do{
    //si hay datos en el buffer de entrada pasarlos a entero
    if (Serial.available() > 0){
      numberToConvert = Serial.parseInt();
    }
  }
  while (numberToConvert == 0);
}
```

---

**ADVERTENCIA:** Para mostrar textos por pantalla en Arduino, es mejor no utilizar tildes en las palabras (usamos numero en lugar de número), ya que de lo contrario pueden no mostrarse correctamente.

---

En la función `getBaseAndConvert()`, debemos mostrar un mensaje de selección de opciones de conversión, leer la entrada que haya seleccionado el usuario, transformar el número a la base indicada y mostrarlo por pantalla. Para leer los datos introducidos, usaremos esta vez `Serial.read()`, que nos devolverá el primer byte de la memoria y deberemos comparar con las opciones de conversión disponibles. Esta manera de leer de la memoria (también llamado *buffer*), además de leer el



byte, lo elimina del *buffer*, de modo que si hacemos otra lectura, leeríamos el siguiente byte. Si el byte leído no es uno de los esperados, se tiene que informar al usuario que el valor introducido no es válido.

```
/*
 * Obtiene la base (b, o, h) y muestra
 * el valor del número en esa base
 */
void getBaseAndConvert() {
    Serial.print("Introduce la base en la que convertir el número ");
    Serial.println(numberToConvert);
    Serial.println("b) Binario");
    Serial.println("o) Octal");
    Serial.println("h) Hexadecimal");
    boolean baseOk = false;
    //el bucle se repetirá mientras no se introduzca un valor apropiado
    do {
        //si hay datos en la entrada leemos el byte
        if (Serial.available() > 0) {
            base = Serial.read();
            //comprobamos el valor del byte
            switch (base) {
                case 'b':
                    Serial.print("En binario es:");
                    Serial.println(numberToConvert, BIN);
                    baseOk = true;
                    break;
                case 'o':
                    Serial.print("En octal es:");
                    Serial.println(numberToConvert, OCT);
                    baseOk = true;
                    break;
                case 'h':
                    Serial.print("En hexadecimal es:");
                    Serial.println(numberToConvert, HEX);
                    baseOk = true;
                    break;
                default:
                    Serial.println("La base introducida no es correcta! Seleccione otra vez");
            }
        }
    } while (baseOk == false);
}
```

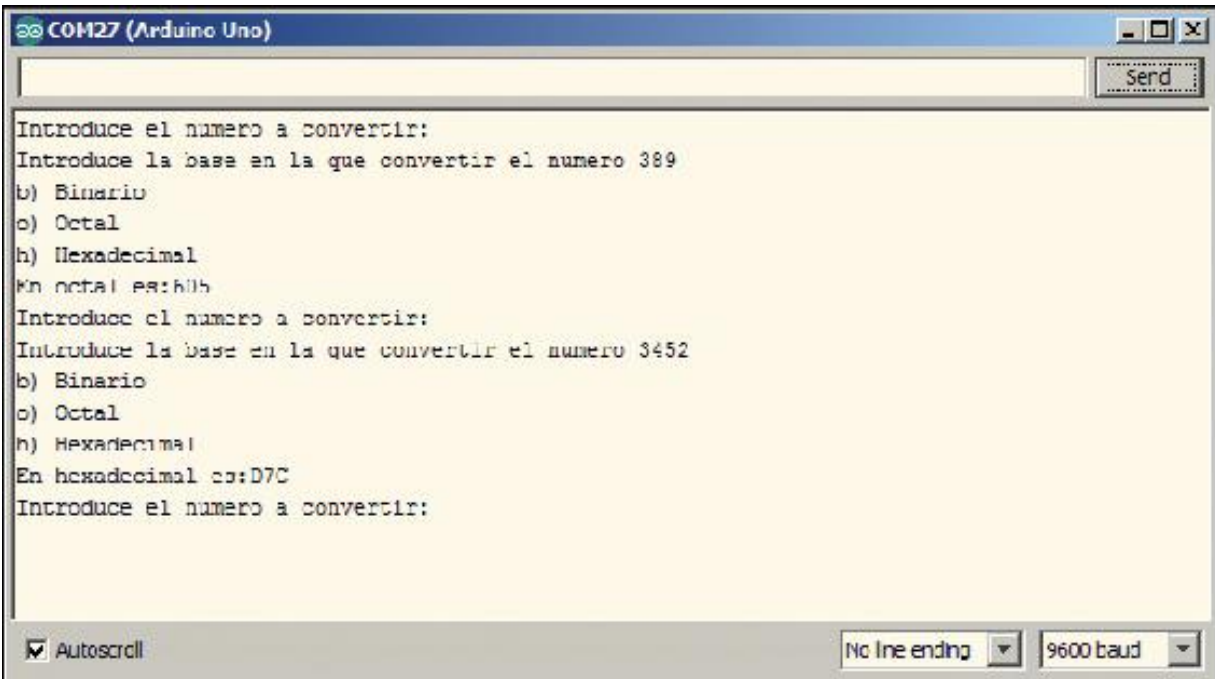


Figura 5.1. Monitor serial con la salida del sketch.

Para usar el programa, necesitamos cargar el *sketch* en la tarjeta y se lanza el monitor serie de modo semejante a como lo hicimos antes. En el monitor aparecerá un mensaje pidiendo que se introduzca un número; en la parte superior de la pantalla se introduce el número a convertir y para enviarlo al puerto serie se debe pulsar sobre el botón **Enviar**. Más adelante se pide que seleccionemos la base a la que transformar el número; del mismo modo se introduce en la caja superior la letra de la base a la que transformar y se obtendrá en la pantalla el número convertido a la base seleccionada. Si en lugar de una letra se introducen varias, el sistema leerá de una en una hasta que una de ellas sea una base válida, mostrando errores para el resto.

El código completo quedaría:

```
unsigned int numberToConvert = 0; // número a convertir
byte base = '\0'; //base a convertir

void setup() {
  // se configura el puerto serie para trabajar a 9600 bps
  Serial.begin(9600);
}
```

```

void loop() {
    //obtener el número a transformar
    getNumber();
    //obtener la base y convertir
    getBaseAndConvert();
    // se inicializan de nuevo las variables para un nuevo ciclo
    numberToConvert = 0;
    base = '\0';
}

/*****
* Obtiene el número a convertir
*****/

void getNumber(){
    Serial.println("Introduce el número a convertir:");
    do{
        //si hay datos en el buffer de entrada pasarlos a entero
        if (Serial.available() > 0){
            numberToConvert = Serial.parseInt();
        }
    }
    while (numberToConvert == 0);
}

/*****
* Obtiene la base (b, o, h) y muestra
* el valor del número en esa base
*****/

void getBaseAndConvert(){
    Serial.print("Introduce la base en la que convertir el número ");
    Serial.println(numberToConvert);
    Serial.println("b) Binario");
    Serial.println("o) Octal");
    Serial.println("h) Hexadecimal");
    boolean baseOk = false;
    //el bucle se repetirá mientras no se introduzca un valor apropiado
    do{
        //si hay datos en la entrada leemos el byte
        if (Serial.available() > 0){
            base = Serial.read();
            //comprobamos el valor del byte
            switch (base){
                case 'b':
                    Serial.print("En binario es:");
                    Serial.println(numberToConvert, BIN);
                    baseOk = true;
                    break;
            }
        }
    }
}

```

```
case 'o':
    Serial.print("En octal es:");
    Serial.println(numberToConvert, OCT);
    baseOk = true;
    break;
case 'h':
    Serial.print("En hexadecimal es:");
    Serial.println(numberToConvert, HEX);
    baseOk = true;
    break;
default:
    Serial.println("La base introducida no es correcta! Seleccione otra vez");
}
}
}
while (baseOk == false);
}
```

# 6

## Entradas y salidas



**En este capítulo aprenderá a:**

- Diferenciar datos analógicos y digitales
- Configurar los pines de la placa para su uso
- Manejar datos analógicos
- Crear un contador de tiempo
- Utilizar pulsadores
- Realizar un comprobador de pilas
- Diseñar un dado electrónico
- Hacer uso de leds

Ya hemos visto que la tarjeta Arduino tiene unos zócalos donde conectar cables para obtener y enviar datos al mundo exterior, son las entradas y salidas de datos. En este capítulo veremos los tipos de configuración que pueden tomar cada uno de estos pines y su utilización en proyectos reales

## **Entradas**

Las entradas pueden ser tanto analógicas como digitales. Dependiendo de la naturaleza de lo que se quiera captar se deberán utilizar un tipo u otro, por ejemplo, si queremos saber si alguien ha pulsado un botón, nos sirve una entrada digital, ya que el botón está pulsado o no lo está, pero si quisiéramos medir una temperatura, queda claro que un valor de 0 o 1 no nos es suficiente para saber los grados que hace.

### **Entradas digitales**

Las entradas digitales son aquellas que pueden detectar los valores 0 y 1, o dicho de otro modo `LOW` y `HIGH`. Como podemos imaginar, realmente no recibimos un 1 del mundo exterior, sino que lo que recibimos es un voltaje, que nosotros interpretaremos como un 1. De manera estándar se toma que 0V representa al 0 y 5V representa al 1, aunque estos voltajes no tienen por qué ser exactos, sino que lo que se hace es que si es cercano al 0V se toma como 0 y si es cercano a 5V se toma como 1, con esto evitamos problemas de pequeñas variaciones en el voltaje debido por ejemplo a interferencias. El valor que marca la diferencia entre 0 y 1 son 3V, es decir voltajes mayores serán tomados como 1 y los menores como 0. Esto no quiere decir que en nuestros circuitos trabajemos cerca de 3V, nada de eso, hay que trabajar lo más alejado posible de ese valor y lo más cercano a 0V y 5V para que interferencias y caídas de voltaje no esperadas no cambien los valores esperados.

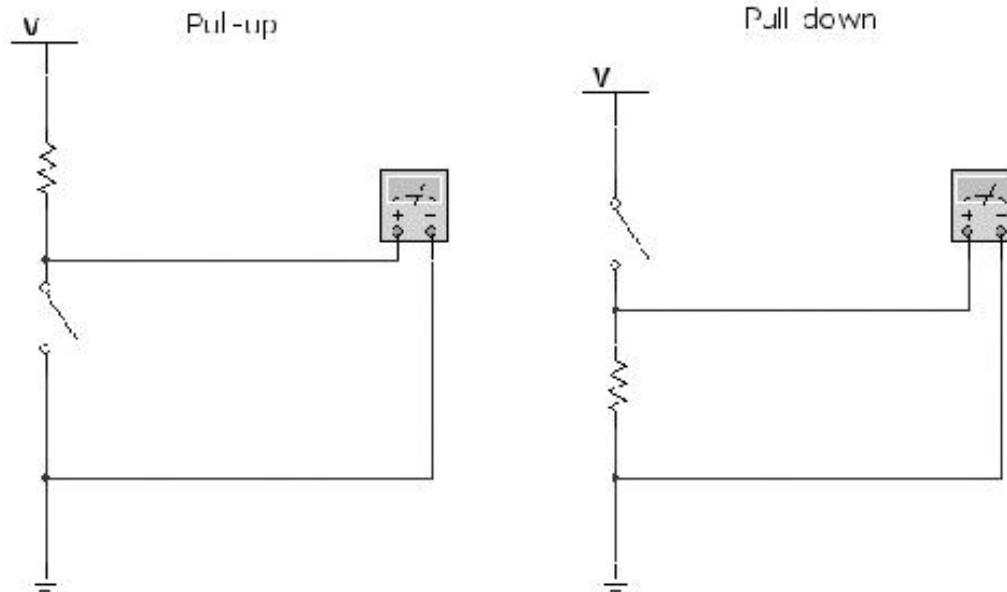
Para trabajar con las entradas digitales, básicamente usaremos dos funciones:

- `pinMode(pin, INPUT)`: que sirve para configurar el pin en modo lectura y donde `pin` es el número de pin digital sobre el que leer.
- `digitalRead(pin)`: para leer el valor de la entrada y donde `pin` es el número de pin digital sobre el que leer. Retornará los valores *HIGH* o *LOW* en función del voltaje recibido.

Para poder realizar las lecturas de las entradas digitales, lo primero que se debe hacer es realizar la configuración del pin o pines de entrada que se quieren leer en la función `setup()`; (realmente no es necesario para la lectura pero es mejor acostumbrarse) y más adelante realizar llamadas a la función `digitalRead(pin)`; con el mismo número de pin que se ha preparado para la lectura.

### Resistencias de pull-up y pull-down

En los montajes con entradas digitales se suelen utilizar unas resistencias externas para obtener los voltajes deseados en las lecturas en caso de que no exista voltaje en la entrada; es decir que si no hay señal, se puede forzar a que tenga un valor determinado *HIGH* o *LOW* mediante unas resistencias convenientemente colocadas, estas resistencias se llaman de *pull-up* o de *pull-down* dependiendo de la situación dentro del montaje. Si cuando estamos en circuito abierto la resistencia está en la rama de 5V se denomina *pull-up* y si se encuentra en la rama de 0V de *pull-down*.



**Figura 6.1.** Resistencias de pull-up y pull-down

La diferencia de trabajar con *pull-up* o *pull-down* está en el valor que toma la entrada en caso de no tener nada conectado a ella; si es *pull-up* tomará el valor HIGH y si es *pull-down* tomará el valor LOW.

En caso de no querer utilizar resistencias de *pull-up* externas, Arduino incorpora unas internas de 20k que podemos activar mediante el código:

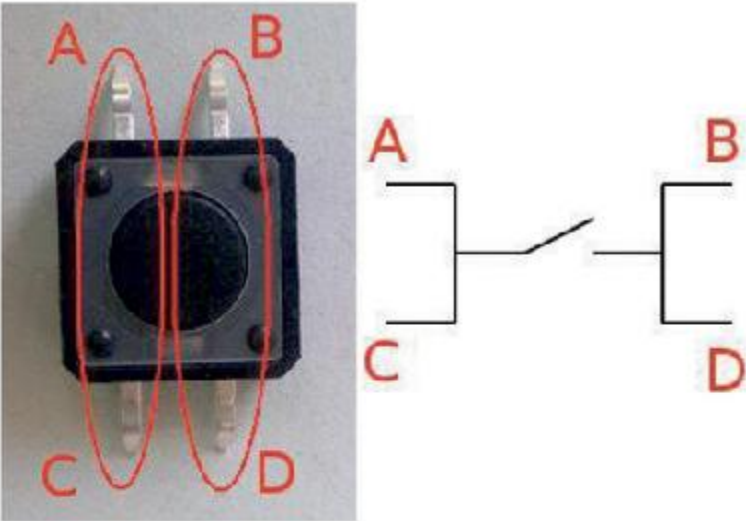
```
pinMode(pin, INPUT); // indica el pin que se quiere poner de entrada
digitalWrite(pin, HIGH); // se activa la resistencia pull-up
```

Vamos a realizar un par de circuitos para ver su funcionamiento. Para ayudarnos en el montaje del circuito, usaremos unos pulsadores electrónicos. Los pulsadores electrónicos tienen 4 patillas o pines que se encuentran unidos de dos en dos cuando no está pulsado y se unen todos ellos cuando se pulsa. Podemos ver el esquema en la figura 6.2 donde se pueden ver las patillas que se encuentran unidas.

Lo que haremos es conectar el pulsador a la entrada 4 de la placa y mostrar en el terminal serie si se está pulsando o no. Además del pulsador, necesitaremos una resistencia de 10KΩ, la *protoboard* y una serie de cables. Los cables que se utilizan en los prototipos suelen ser de muchos colores, facilitando seguir “la pista” a cada rama del circuito y aunque

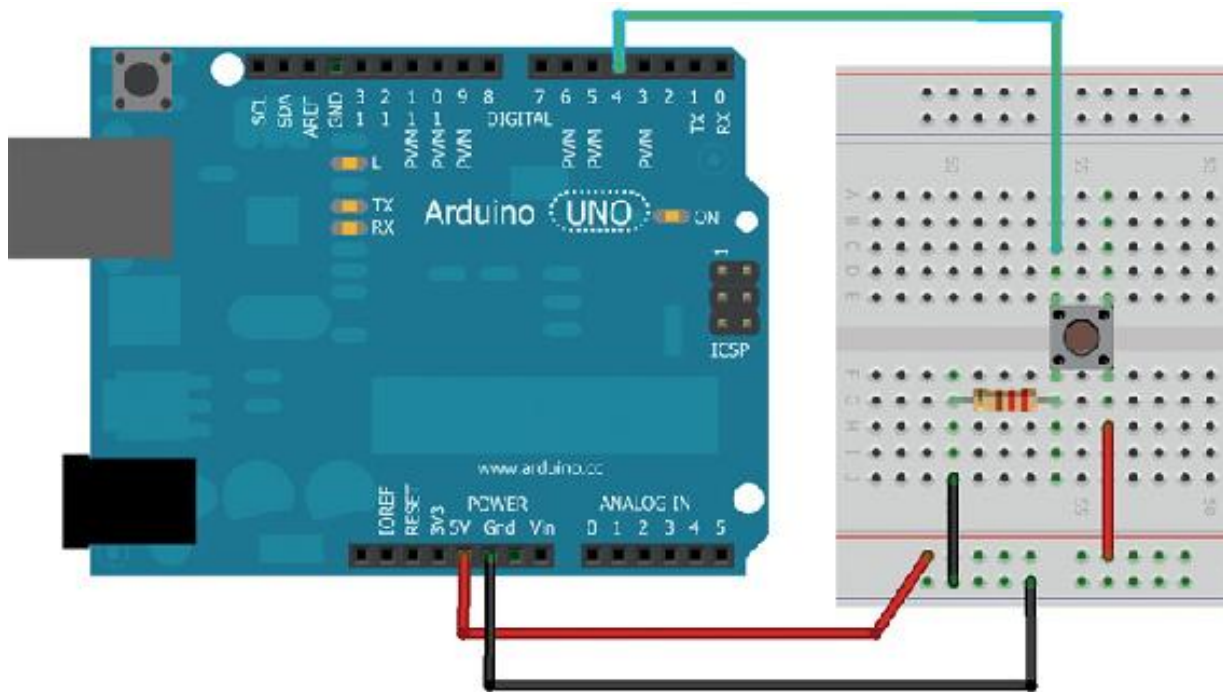


podemos utilizar cualquier cable en cualquier sitio, en circuitos complejos es mejor ordenar las conexiones por colores, por ejemplo los rojos para la alimentación positiva, los negros para tierra o 0V...



**Figura 6.2.** Pulsadores electrónicos

El primer montaje lo realizaremos con una resistencia *pull-down*. Con esta configuración, en caso de circuito abierto (no se está pulsando el botón), la lectura será de 0V o LOW. El montaje del circuito lo podemos ver en la figura 6.3.



**Figura 6.3.** Circuito con resistencia pull-down

Lo que haremos en el bloque de configuración será preparar tanto el pin de entrada como la conexión serie para poder mostrar mensajes y en el bucle principal, leer la entrada y dependiendo del valor leído mostrar un mensaje u otro. La comprobación de la lectura se realizará mediante una estructura `if` y comparando con el valor `HIGH` y claro está que si el valor no es `HIGH`, será `LOW`, mostrando que está libre.

```
const byte inputPin = 4; // pin para el botón
void setup() {
  pinMode(inputPin, INPUT); // se declara como pin de entrada
  Serial.begin(9600);
}
void loop(){
```

```

int val = digitalRead(inputPin); // lectura del valor
if (val == HIGH) {
Serial.println("Pulsado");// está pulsado
}
else{
Serial.println("Libre!!!!");// no está pulsado
}
delay(500);//se detiene el programa medio segundo antes de un nuevo ciclo.
}

```

Al probar el programa, podemos ver en el terminal los textos “**Pulsado**” o “**Libre!!!!**” dependiendo de si mantenemos apretado o no el pulsador.

Si cambiamos la configuración para trabajar con *pull-up*, cuando no se aprieta el pulsador, tendremos el circuito abierto y en la entrada tendremos 5V que es un HIGH. El circuito en pull-up sería el que se muestra en la figura

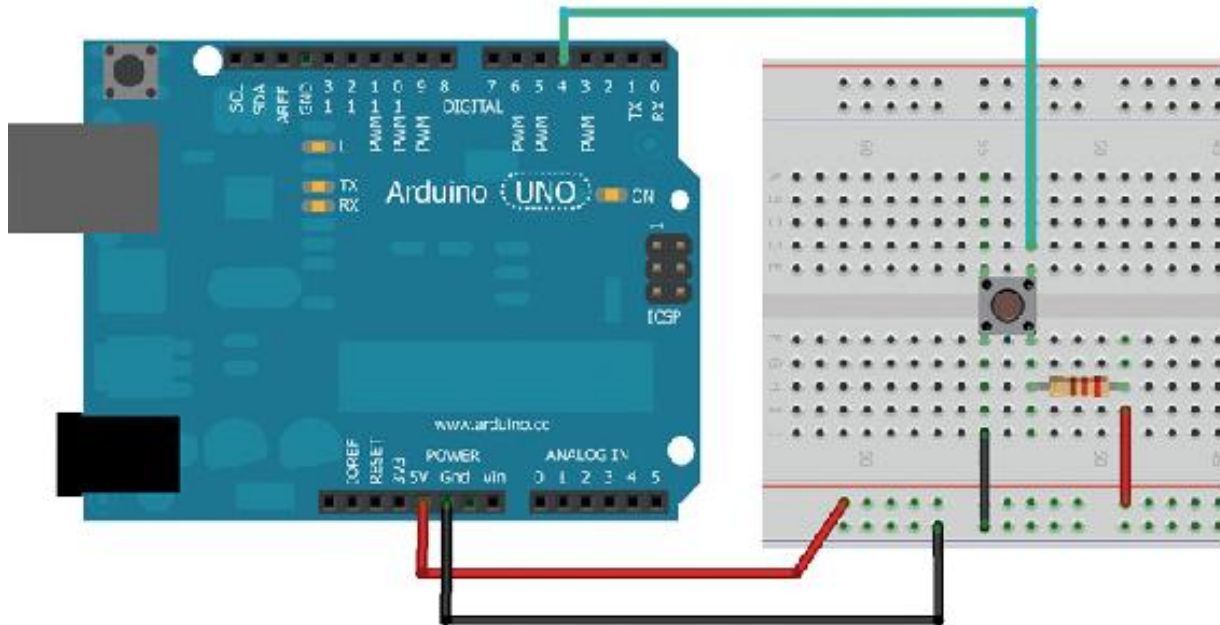


Figura 6.4. Circuito con resistencia pull-up

Si se trabaja con esta configuración tendremos que tener en cuenta que el HIGH indica que el pulsador no está apretado, es decir habría que cambiar

```

if (val == HIGH) {
por

```

```
if (val == LOW) {
```

Las placas Arduino permiten hacer este tipo de montaje sin necesidad de resistencias externas para facilitarnos el trabajo, de modo que podemos configurar la entrada para trabajar con resistencias *pull-up* sin elementos adicionales, es decir, que en circuito abierto tendremos una lectura de HIGH. El montaje sin resistencias externas quedaría como el mostrado en la figura 6.5

Para poder trabajar con estas resistencias internas, tenemos que configurar la entrada escribiendo en ella HIGH en el momento de la configuración:

```
void setup() {  
  pinMode(4, INPUT); // botón de comienzo  
  digitalWrite(4, HIGH); // se activa la resistencia de pull-up  
}
```

Podemos trabajar con tantas entradas y salidas como queramos hasta un máximo definido por lo que ofrezca la placa que utilicemos. Vamos ahora a hacer un nuevo *sketch* en el que trabajaremos con dos botones. Lo que vamos a hacer en este caso es que al pulsar uno de ellos comenzaremos a contar tiempo y cuando pulsemos el otro dejaremos de contar y mostraremos el tiempo transcurrido entre las dos pulsaciones. El funcionamiento básico será el siguiente: al pulsar el botón de comenzar guardaremos el tiempo en el que se ha hecho la pulsación mediante la llamada a la función estándar `millis()` que devuelve el momento actual en milisegundos, cuando se pulse el botón de finalizar, volveremos a llamar a esta función y restaremos los resultados, de modo que obtendremos la diferencia en milisegundos entre las dos pulsaciones. Hay unos casos especiales a tener en cuenta y son qué hacer si se pulsa al botón de empezar cuando ya está contando o si se pulsa al de finalizar si no está contando.

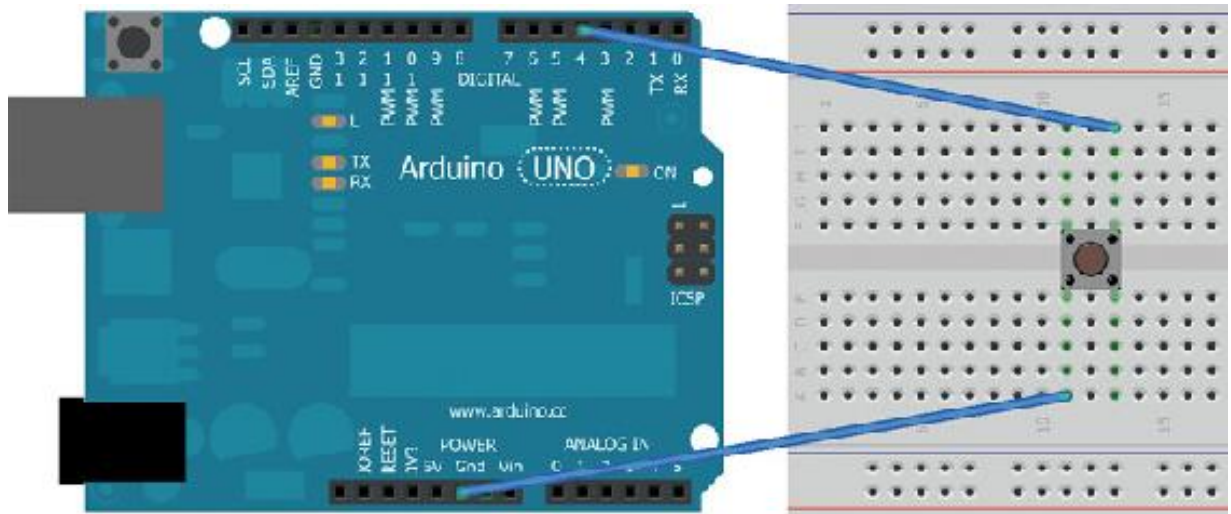


Figura 6.5. Circuito con resistencia pull-up interna

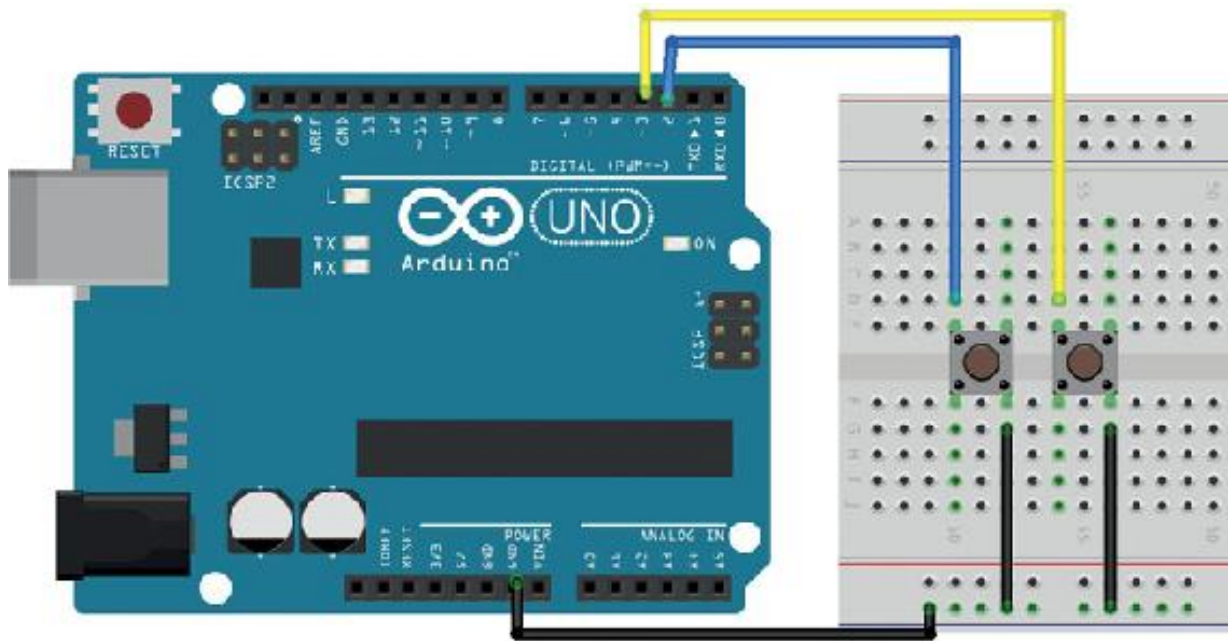


Figura 6.6. Reloj contador

El montaje sería como el mostrado en la figura 6.6 y el *sketch*:

```

unsigned long start, finished, elapsed;
boolean running;

void setup() {
  Serial.begin(9600);
  pinMode(2, INPUT); // botón de comienzo
  digitalWrite(2, HIGH); // se activa la resistencia de pull-up

```

```
pinMode(3, INPUT); // botón de finalización
digitalWrite(3, HIGH); // se activa la resistencia de pull-up
Serial.println("Pulse 1 para comenzar y 2 para finalizar");
running = false;
}
```

```
void displayResult() {
float h, m, s, ms;
unsigned long over;
elapsed = finished-start;
h = int(elapsed / 3600000);
over = elapsed % 3600000;
m = int(over / 60000);
over = over % 60000;
s = int(over / 1000);
ms = over % 1000;
Serial.print("Tiempo transcurrido en milis: ");
Serial.println(elapsed);
Serial.print("Tiempo transcurrido: ");
Serial.print(h, 0);
Serial.print("h ");
Serial.print(m, 0);
Serial.print("m ");
Serial.print(s, 0);
Serial.print("s ");
Serial.print(ms, 0);
Serial.println("ms");
Serial.println();
}
```

```
void loop() {
if (digitalRead(2) == LOW) { //estamos en pull-up
if (running){
Serial.println("Ya estamos contando...");
}else{
running = true;
start = millis();
delay(200); // por seguridad
Serial.println("Comenzamos...");
}
}

if (digitalRead(3) == LOW){ // estamos en pull-up
if (running){

running = false;
finished = millis();
delay(200); // por seguridad
```

```

displayResult();
}
else{
Serial.println("Pulse comenzar antes de finalizar...");
}
}
}
}

```

Lo primero que hacemos en el *sketch* es crear variables para mantener los tiempos de pulsado de inicio y fin, de contar el tiempo y otra más para guardar la diferencia entre los tiempos (la variable `elapsed`); también creamos la variable `running` que será *true* cuando esté contando el tiempo y *false* cuando no esté contando.

En el `setup()` se tiene que configurar el terminal serie y los botones para trabajar con las salidas 2 y 3 que usarán resistencias internas de *pull-up*.

La función `displayResult()` se encargará de pasar de milisegundos a algo más entendible, mostrando horas, minutos, segundos y milisegundos; la llamaremos cuando estando contando tiempo, el usuario pulse el botón 2.

Dentro del bucle principal `loop()` es donde se controla la lectura de las entradas. Si la lectura en el pin 2 es `LOW` significa que han pulsado comenzar, por lo que comprobaremos si está ya contando o no, si está ya contando mostramos un mensaje en pantalla y si no está contando, comenzamos a contar, lo que significa que ponemos la variable `running` a *true*, guardamos el tiempo actual en milisegundos con `start = millis()`, mostramos un mensaje en pantalla indicando que comenzamos a contar. Si lo que se pulsa es el botón 2 que está en la entrada 3, en ésta tendremos el `LOW`, con lo que tenemos que poner la variable `running` a *false*, guardamos el tiempo actual en milisegundos con `finished = millis()` y mostramos el resultado llamando a `displayResult()`; en este caso se tiene en cuenta también si pulsamos el botón 2 y no se está contando el tiempo, para indicar al usuario que tiene que comenzar antes a contar.

Si probamos el programa iremos viendo en la salida algo semejante a:

Pulse 1 para comenzar y 2 para finalizar

Comenzamos...

Tiempo transcurrido en milis: 27781

Tiempo transcurrido: 0h 0m 27s 781ms

Comenzamos...

Tiempo transcurrido en milis: 2919

Tiempo transcurrido: 0h 0m 2s 919ms

Pulse comenzar antes de finalizar...

Pulse comenzar antes de finalizar...

Pulse comenzar antes de finalizar...

En la salida anterior podemos ver que si pulsamos el botón 2 antes del botón 1 nos dice que pulsemos al de comenzar antes que finalizar... pero nos lo enseña varias veces; esto es porque se ejecuta varias veces el `loop()` principal mientras tenemos pulsado el botón (por muy rápidos que seamos). Estas repeticiones se pueden controlar usando una variable que indique si la última vez que se consultó el estado del botón es igual que el estado actual o no, pero en este caso no la hemos incluido en el ejemplo.

## **Entradas analógicas**

Cuando hablamos de entradas analógicas, los valores pueden ser múltiples no solo 0 o 1. Podríamos decir que las entradas digitales son como un interruptor que enciende o apaga una bombilla mientras que las analógicas son esos interruptores que te dejan seleccionar la intensidad con la que ilumina la bombilla. Cuando Arduino recibe una entrada analógica lo hará con un voltaje entre 0 y 5 voltios y este valor la propia tarjeta lo transformará en valores entre 0 y 1023 dependiendo del valor recibido y será con el valor que trabajaremos nosotros. Para obtener el dato analógico de la entrada se utiliza la función `analogRead(pin)`; donde `pin` es el número del pin analógico que se debe leer, devolviendo la llamada el valor obtenido entre 0 y 1023. Como sabemos que si hay en la entrada 5V Arduino nos daría el valor 1023, para saber en cualquier momento el valor de la entrada podemos aplicar una regla de tres, por ejemplo si en el pin de entrada existen 2.3 voltios, el valor recibido sería  $2.3V * 1023 / 5V = 471$  o que si leemos 471 en nuestro programa podemos obtener el voltaje



mediante la fórmula  $5V \cdot 471 / 1023 = 2.3V$ . A partir de este valor es el programa quien tiene que interpretarlo, porque dependiendo de quién genere esa entrada será un dato u otro, así si es un termómetro quien lo genera será entonces grados centígrados o Fahrenheit y si es un detector de luz serán lux, además depende de cada fabricante y modelo, esos 2.3V se tienen que interpretar como 20°C o 38°C, para ello deberemos seguir las instrucciones que nos dé el fabricante, más adelante veremos un ejemplo.

Los pines analógicos vienen marcados en la tarjeta con la etiqueta **ANALOG IN**.



**Figura 6.7.** Potenciómetros

Para ver cómo funcionan las entradas analógicas usaremos una resistencia variable también llamada potenciómetro. Esta resistencia tiene la particularidad de que el valor de su resistencia puede ser variado en cualquier momento.

El potenciómetro normalmente tiene tres patillas y la mayor resistencia se da entre las entradas más externas, obteniendo la resistencia variable a partir de la patilla intermedia. Si un potenciómetro tiene de valor 10KΩ (la K significa mil, es decir 10.000 ohmios) de valor nominal (valor máximo), en caso de medir la resistencia entre los pines externos obtendremos ese valor, mientras que si lo medimos entre cualquiera de las patillas externas y la intermedia, obtendremos un valor tal, que si

medimos entre la intermedia y la otra patilla externa, la suma daría la resistencia nominal, es decir los 10KΩ. Según se mueve el mando del potenciómetro modificaremos esas lecturas.

$$V_1 = I * (R_1 + R_2) \Rightarrow I = \frac{V_1}{(R_1 + R_2)}$$

$$V_{salida} = I * R_2 = V_1 * \frac{R_2}{(R_1 + R_2)}$$

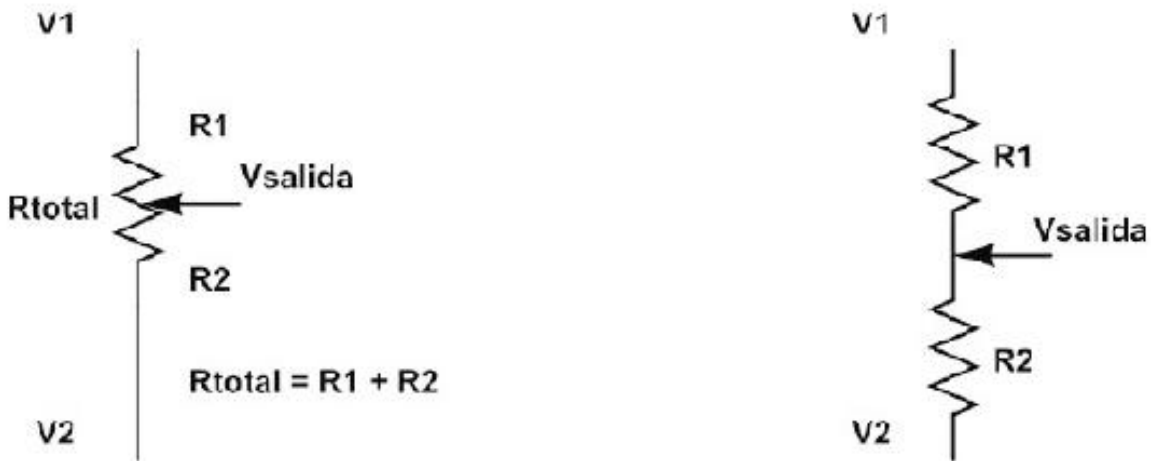


Figura 6.8. Divisor de tensión

El potenciómetro trabaja como lo que se llama divisor de tensión, de modo que la caída de voltaje total se realiza entre sus dos patillas externas y es igual a la caída de tensión o voltaje entre una de las patillas externas y la interna más la caída de tensión entre la patilla interna y la otra patilla externa. Solo para los más matemáticos, en la figura 6.8 tenemos la fórmula del voltaje que consume cada parte del potenciómetro.

Para ver cómo funcionan tanto potenciómetros como entradas analógicas vamos a hacer un pequeño circuito donde mostraremos en el monitor serie la caída de tensión de un potenciómetro.

El circuito a montar es el de la figura 6.9.

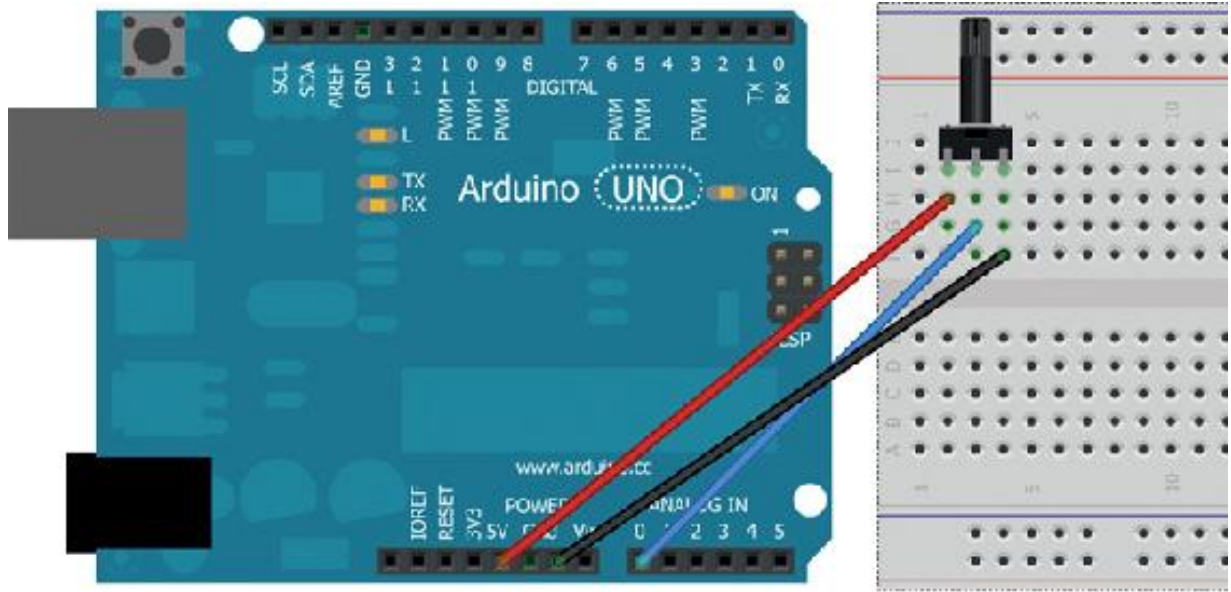


Figura 6.9. Montaje con potenciómetros

El *sketch* simplemente debe leer la entrada correspondiente a la patilla central, donde obtendremos la caída de tensión que se produce:

```
int analogValue = 0;
float voltage = 0;

void setup() {
  //preparamos el terminal serie
  Serial.begin(9600);
}

void loop() {
  analogValue = analogRead(0);
  voltage = 0.0048*analogValue;
  Serial.print("Valor analogico: ");
  Serial.println(analogValue);

  Serial.print("Voltaje: ");
  Serial.print(voltage);
  Serial.println("V");
  delay(1000);
}
```

Usaremos dos variables, una `float` que puede tener decimales para el voltaje y otra de tipo `int` que tendrá un número entero que es el devuelto por Arduino. Lo que haremos será transformar el valor que devuelve

Arduino (el entero) en los voltios reales que tenemos en la entrada mediante una regla de tres; sabiendo que si hay 5V en la entrada tendríamos una lectura de 1023, pues si tenemos una lectura de 274, el valor real de los voltios sería  $274 * 5V / 1023$ , es decir multiplicar el valor obtenido por  $5V/1023$  que es el valor mágico 0.0048. Una vez tenemos el valor, lo mostramos en pantalla. Si ejecutamos el programa y vamos moviendo el potenciómetro tendremos una salida semejante a:

```
Valor analogico: 0
Voltaje: 0.00V
Valor analogico: 274
Voltaje: 1.32V
Valor analogico: 513
Voltaje: 2.46V
Valor analogico: 802
Voltaje: 3.85V
Valor analogico: 1023
Voltaje: 4.95V
```

Pues ahora, con unas pocas modificaciones en el circuito, podemos tener un comprobador de pilas de hasta 5 voltios. Si utilizamos el montaje de la figura 6.10, en la salida del terminal serie podemos ver los valores de voltaje de cada pila que probemos, pudiendo ver si esta gastada o no. Por ejemplo para una pila RL6 típica, tendremos una salida semejante a:

```
Valor analogico: 317
Voltaje: 1.52V
Valor analogico: 317
Voltaje: 1.52V
Valor analogico: 316
Voltaje: 1.52V
```

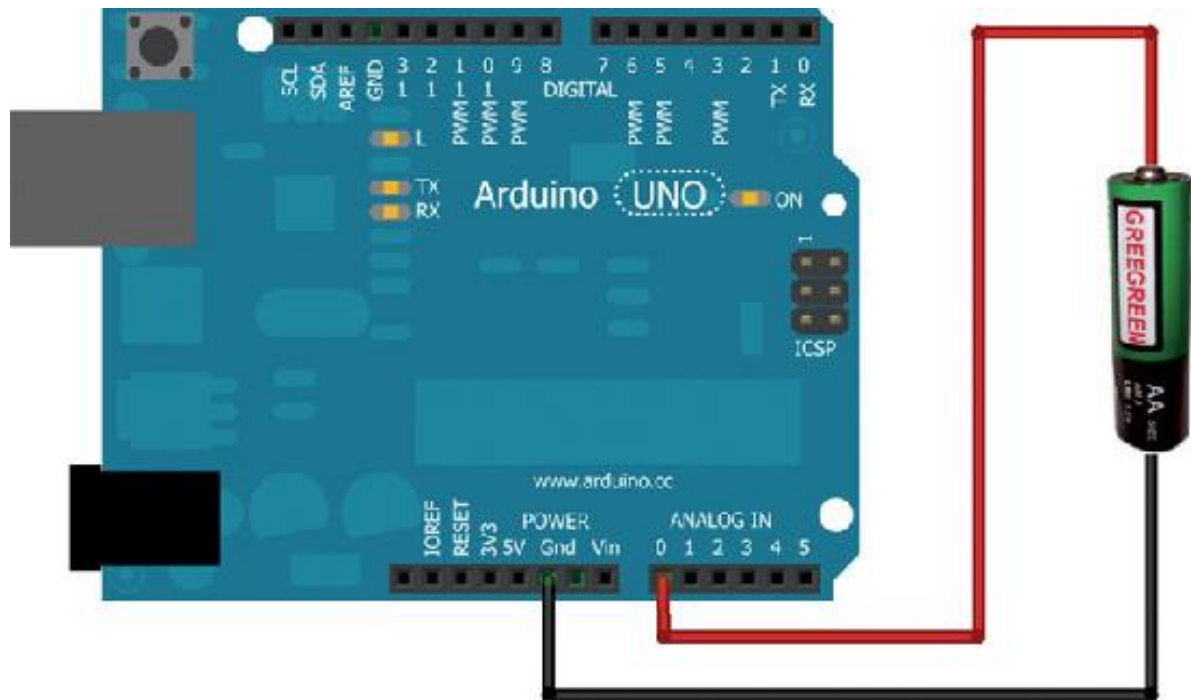


Figura 6.10. Comprobador de pilas

## Salidas

Las salidas, al igual que las entradas, pueden ser analógicas como digitales. Las salidas nos servirán para indicar a los actuadores que deben realizar alguna acción que dependiendo del actuador, puede ser encenderse en caso de ser un led o girar si es un motor.

### Salidas digitales

Las salidas digitales son aquellas que pueden tomar los valores LOW y HIGH o dicho de otro modo 0 y 5 voltios. Por ejemplo para encender una bombilla (un led) usaremos los 5V que nos ofrecen las salidas de Arduino. Teniendo en cuenta el voltaje que se obtendrá a la salida, debemos adecuar los circuitos a los actuadores que tengamos que utilizar, porque a lo mejor

necesitamos menos de 5V para activar el elemento que tengamos conectado.

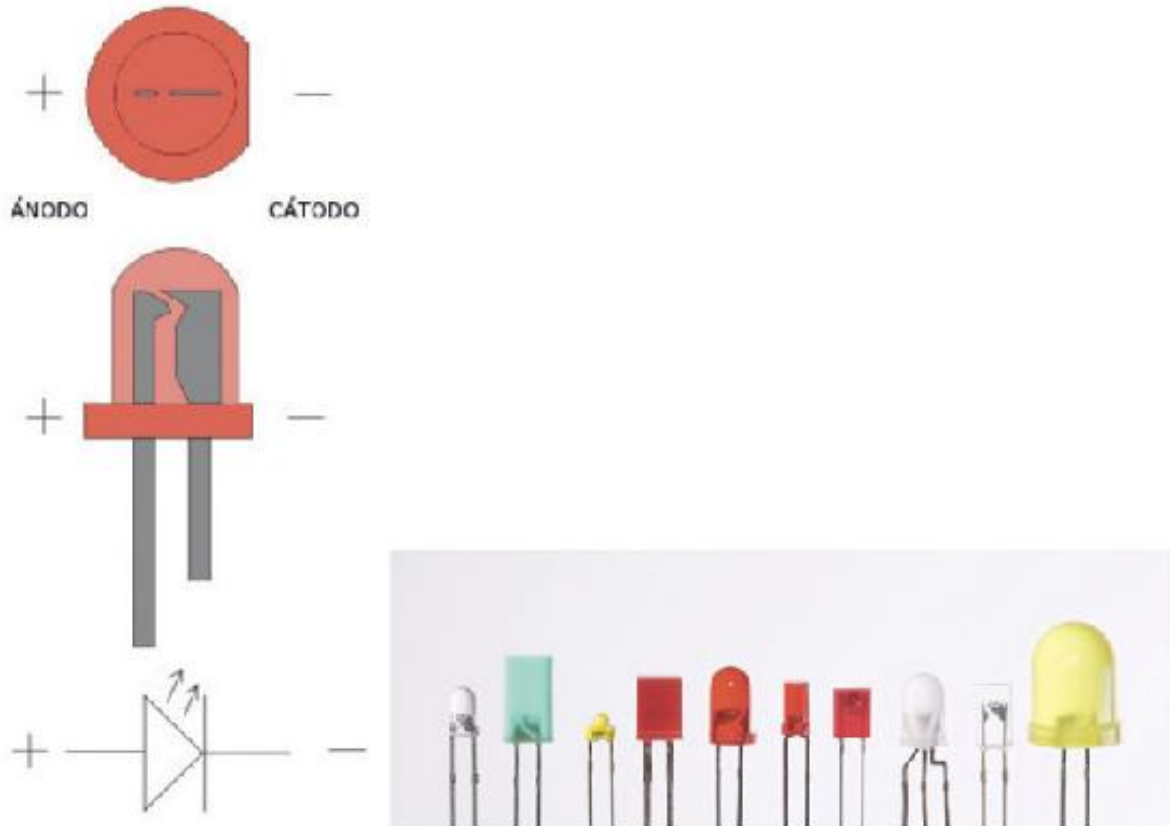
Para trabajar con las salidas digitales, de manera semejante a las entradas, usaremos básicamente dos funciones:

- `pinMode(pin, OUTPUT)`: que sirve para configurar el pin en modo escritura y donde `pin` es el número de pin digital sobre el que escribir.
- `digitalWrite(pin, value)`: La variable `value` es el valor que se quiere dar a la salida en el pin indicado por el parámetro `pin`.

Para ver cómo funcionan estas salidas, vamos a reutilizar el sketch de ejemplo en el que encendíamos el led (el ejemplo Archivo > Ejemplos > 01.Basicos > Blink) pero en este caso encenderemos un led externo (figura 6.11).

En la figura 6.11 podemos ver cómo son los led físicamente, estos dispositivos tienen dos pines, uno llamado ánodo y otro llamado cátodo. Para que funcione correctamente, el ánodo debe estar conectado a mayor tensión que el cátodo. Como es muy importante polarizar de manera correcta el led, tenemos que distinguir entre el ánodo y el cátodo y para ello los led tienen varias marcas; la más clara y que se da en todos los leds es que la pata del cátodo (recordemos que es la que debe ir al negativo) es más corta que la del ánodo. A veces esto no es suficiente porque hemos cortado los pines o por otras causas, por lo que podemos fijarnos en otras características, como que muchos leds tienen en el recubrimiento epoxi (la caperuza) una marca en la pata del cátodo, bien puede ser un plano o un rebaje; por último si nos fijamos en el interior del led, la placa que se sitúa por encima es la que corresponde al cátodo. En caso de no tener muy claro si es el ánodo o el cátodo por estar muy dañado exteriormente podemos utilizar un polímetro y en la posición de medición de resistencia (Óhmetro) y en unidades lo más bajas posible, tendremos entre los bornes del polímetro una tensión muy pequeña que podemos aplicar directamente al led, si se ilumina será que el borne positivo del polímetro está

conectado al ánodo y si no se ilumina puede ser que esté conectado al cátodo o que esté fundido.



**Figura 6.11.** Aspecto y representación de los leds

Para que funcione un diodo led (o simplemente led), normalmente se usan corrientes de entre 10mA y 25mA (mili amperios). Para conseguir esta intensidad y no fundir el led, debemos poner una resistencia de modo que se limite la intensidad que recorre la rama. Existen diversos colores de leds y dependiendo del color tienen distintas caídas de tensión que tendremos que tener en cuenta para el cálculo de la resistencia.

**Tabla 6.1.** Caídas de tensión en distintos diodos led.

Color	Caída en voltios
Rojo	1,6–2,03

Naranja	2.03–2.10
Amarillo	2,1–2,3
Verde	1,9–3,7
Azul	2,47–3,7
Blanco	3,5
Infrarrojo	< 1,63
Ultravioleta	3,1–4,3

Mediante la fórmula de la ley de Ohm que conocimos en capítulos anteriores, sabiendo que la salida del puerto Arduino son 5V y que usaremos un led rojo que son más o menos 2V de caída de tensión, podemos calcular fácilmente la resistencia a utilizar.

$$R = (V_{\text{arduino}} - V_{\text{led}}) / I = 5V - 2V / 20\text{mA} = 150\Omega.$$

Esta resistencia es un valor aproximado y no hace falta que sea exactamente este valor; para este circuito usaremos un led rojo y una resistencia de 220Ω que son más fáciles de encontrar.

El circuito quedaría como en la figura 6.12.



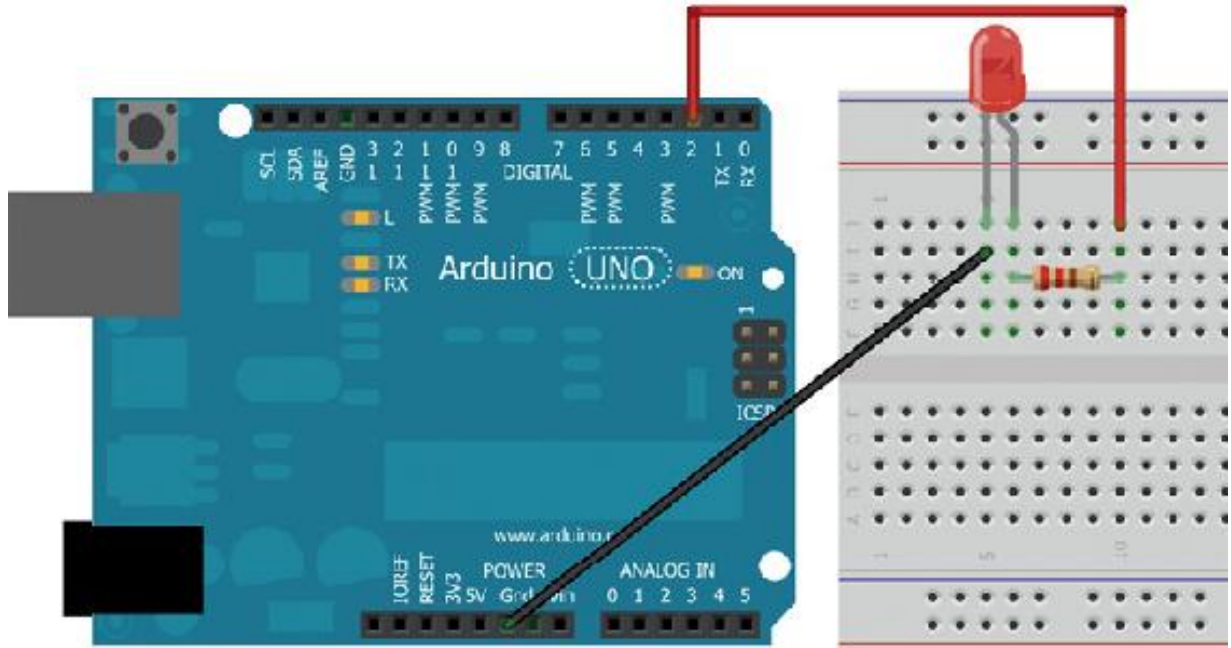


Figura 6.12. Led conectado a salida digital

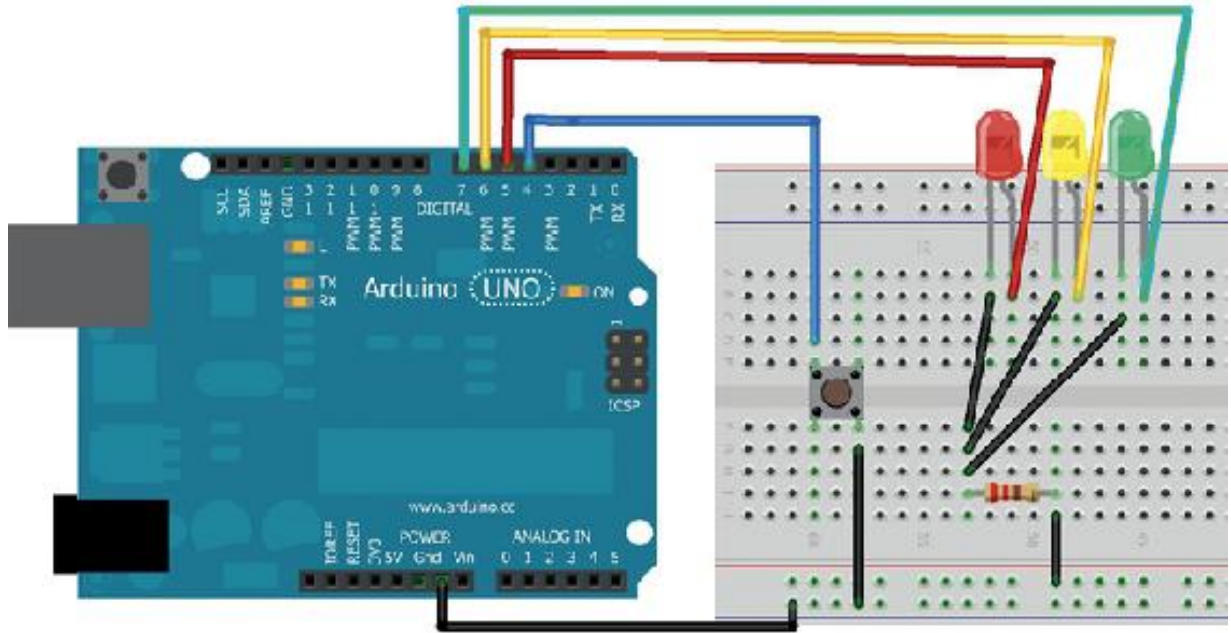
El código del *sketch* se debe encargar simplemente de poner en HIGH y en LOW de manera intermitente la salida digital, de modo que vaya encendiendo y apagando el led. Usaremos el mismo código que en el ejemplo *blink* visto anteriormente, solo que cambiando el pin 13 por el pin 2.

```
void setup() {  
  pinMode(2, OUTPUT);  
}  
  
void loop() {  
  digitalWrite(2, HIGH); // encendemos el led  
  delay(1000); // esperamos  
  digitalWrite(2, LOW); // apagamos el led  
  delay(1000); // esperamos  
}
```

Siguiendo con los leds, vamos a subir la cantidad de ellos y haremos un semáforo con un pulsador para peatones. Utilizaremos tres leds (si es posible, uno rojo, uno amarillo y uno verde para dar más realismo) externos para simular un semáforo de coches, al que añadiremos un pulsador para peatones que hará que se ponga el semáforo de coches en

rojo y encienda el verde en el semáforo de peatones, que en nuestro caso será el led de la placa Arduino, también necesitaremos una resistencia de  $220\Omega$ , que usaremos para todos los leds.

El montaje quedaría como en la figura 6.13.



**Figura 6.13.** Circuito de semáforo

A la hora de realizar el sketch, podríamos tener la tentación de hacer como en el ejemplo *Blink* y poner una llamada a `delay()` de manera que pusiera el semáforo en rojo, esperara por ejemplo 5 segundos, luego lo pusiera en verde, otra espera de unos segundos pasar de nuevo a amarillo y así sucesivamente. El problema que tiene la función `delay()` es que durante el tiempo que le digamos, no hace nada, y este no hacer nada incluye no leer las entradas, es decir el peatón pulsaría el botón pero no le haría caso el semáforo porque no se estaría leyendo la entrada.

En el *sketch* que veremos a continuación, vamos a evitar este caso y dejaremos que el peatón pueda en cualquier momento interrumpir el paso de coches.

```
const byte buttonPin = 4; // pin para el botón
const byte redLedPin = 5; // pin para led rojo
const byte yellowLedPin = 6; // pin para led amarillo
```

```

const byte greenLedPin = 7; // pin para led verde
const byte personLedPin = 13; // pin para led peatones

int lightStatus = 0;

void setup() {
pinMode(personLedPin, OUTPUT);
pinMode(redLedPin, OUTPUT);
pinMode(yellowLedPin, OUTPUT);
pinMode(greenLedPin, OUTPUT);
//config para el pulsador
pinMode(buttonPin, INPUT);
digitalWrite(buttonPin, HIGH); // se activa la resistencia de pull-up
}

void loop() {
if (!digitalRead(buttonPin)) { // si se pulsa
lightStatus = 0; //se fuerza a rojo para coches
}

//dependiendo de la luz que se debe mostrar tiene distintos parámetros la llamada
switch (lightStatus) {
case 0: //rojo
showLight(redLedPin, 3000); //3 segundos
break;

case 1: // verde
showLight(greenLedPin, 2000); //2 segundos
break;
case 2: // amarillo
showLight(yellowLedPin, 1000); //1 segundo
break;
}
}

/**
 * Enciende el led que se le indique.
 * Los parámetros son el led a encender y el tiempo que debe estar encendido
 */
void showLight(int ledPin, int maxTime) {
static int lastPin = 0; //último led encendido
static unsigned long startTime = 0; //tiempo desde que se encendió el led
/* Si el led actualmente encendido no concuerda con el que se debe encender
 * entonces apagar el led encendido, encender el nuevo y poner el contador de tiempo a 0
 */

if (ledPin != lastPin) {
// mirar el semaforo de peatones
if (ledPin == redLedPin) {

```

```

//si es rojo para coches, es verde para peatones
digitalWrite(personLedPin, HIGH);
}else {
    digitalWrite(personLedPin, LOW);
    }
    startTime = millis();
    digitalWrite(lastPin, LOW);
    lastPin = ledPin;
    digitalWrite(ledPin, HIGH);
    delay(5); // para garantizar la estabilidad
}

    unsigned long elapsedTime = millis()-startTime; //tiempo pasado desde que se
encendió el led
    if ((elapsedTime) > maxTime ) { // si es mayor que lo esperado, pasar al siguiente
ciclo de led
        lightStatus++;
        if (lightStatus > 2) lightStatus = 0; // solo hay 3 led, comenzamos de nuevo por
el 0
    }
}

```

Lo primero que hacemos es preparar unas constantes que representarán los distintos pines de la placa que usaremos, además también crearemos una variable para mantener el estado del semáforo. En cada inicio de la función `loop()` se comprobará si el usuario ha pulsado el botón de paso para peatones o no; esto se hace con la llamada `if (!digitalRead(buttonPin))`, que se encarga en la misma línea de leer la entrada del botón, negarla (estamos con *pull-up*) y si es positiva, quiere decir que está pulsado el botón, entonces ejecuta la sentencia dentro del `if`, que lo que hace es forzar el `lightStatus` a 0, es decir a rojo para los coches y verde a los peatones. El `switch` se encarga de saber que led tiene que encender, 0 para rojo, 1 para verde y 2 para amarillo; lo hemos hecho así porque es la manera de funcionar un semáforo, de rojo a verde, de verde a amarillo y de amarillo a rojo, y así para saber que led tenemos que encender, podemos usar una variable que simplemente se incremente. La función `showLight()` es la que se encarga realmente de encender cada led. Si el último led encendido no es el mismo que el que tiene que mostrar, entonces apagará el que esté encendido y encenderá el que se espera. En este cambio de encendidos de led, hay que tener en cuenta si el que se

enciende es el rojo, ya que hay que encender también el verde de los peatones (el correspondiente a la placa en nuestro caso). Al final de la función, se resta el tiempo actual al tiempo que hace que está encendido el led, si es mayor que el tiempo que tenía que estar encendido (tiempo que se pasa por parámetro) entonces sumamos 1 a la variable `lightStatus` para que el semáforo cambie de luz, teniendo en cuenta que si vale 3, hay que hacer que valga 0, ya que no tenemos más que 3 luces. Al definir las variables como `static`, lo que hacemos es que entre llamadas a la función mantienen su valor, de lo contrario, cada vez que se llamara a la función perderían su valor:

```
static int lastPin = 0; //último led encendido
static unsigned long startTime = 0; //tiempo desde que se encendió el led
```

Aprovechando que ya sabemos cómo encender más de un led, vamos a hacer un dado electrónico con seis leds. Este dado constará de un pulsador, que al ser apretado “lanzaré” un dado, encendiendo y apagando cada uno de los seis leds y dejando fijo al final el resultado.

Para el montaje necesitaremos 6 leds, 1 resistencia de  $220\Omega$  o similar y un pulsador y su colocación es la mostrada en la figura 6.14. En este caso también hemos usado una misma resistencia para todos los leds, aunque se podría usar uno para cada uno y dado que hay bastantes conexiones, hay que tener cuidado con los pines que se utilizan para cada una de las conexiones.

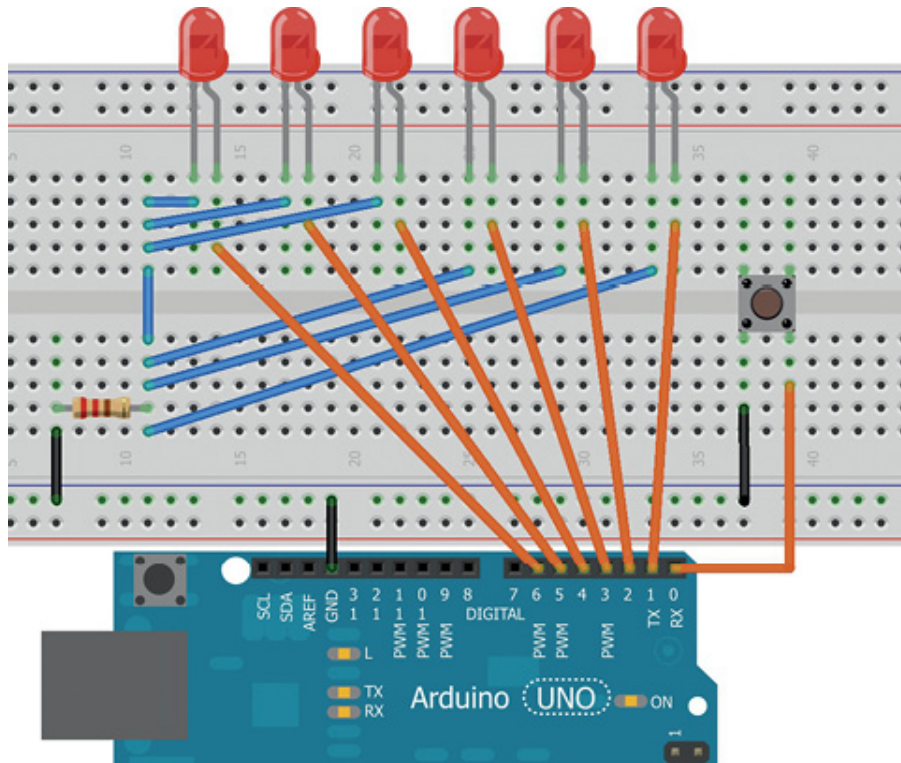


Figura 6.14. Circuito para un dado

El *sketch* quedaría:

```
void setup(){
  randomSeed(analogRead(0)); // generamos la semilla aleatoria
  for ( int i = 1 ; i < 7 ; i++ ){ // los leds serán las salidas de 1 a 6
    pinMode(i, OUTPUT);
  }
}

void loop(){
  int i;
  // bucle sobre los leds para encender y apagar aleatoriamente
  for ( i = 0 ; i < 100 ; i++ ){
    randomLED(50);
  }
  // más despacio
  for ( i = 1 ; i <= 10 ; i++ ){
    randomLED(i * 100);
  }
  // paramos el led que esté encendido
  randomLED(0);
}

/** Enciende y apaga los leds de manera aleatoria pausando
los milisegundos indicados por el parámetro
**/
```

```

void randomLED(int del){
  int r = random(1, 7); // número aleatorio de 1 a 6
  digitalWrite(r, HIGH); // ponemos a 1 la salida del led aleatorio
  if (del > 0){
    delay(del); // pausa
  }
  else if (del == 0){
    do {
// pausamos hasta que haya pulsación
}while (digitalRead(0));
  }
  digitalWrite(r, LOW); // apagamos el led aleatorio
}

```

Para generar números aleatorios, se necesita un número que se llama semilla que servirá para, a partir de él, generar nuevos números aleatorios; este número aleatorio lo tomaremos leyendo la entrada analógica 0, donde no tenemos nada conectado, así que el valor que leerá será bastante al azar, ya que leerá lo que se llama ruido electrónico, que depende de los aparatos electrónicos que tengamos alrededor. Este número se tiene que asignar como semilla, tanto la lectura como la asignación lo hacemos mediante las llamadas `randomSeed(analogRead(0))`. Como parte del `setup()` también indicamos que los pines del 1 al 6 serán de salida. El bucle `loop()` llamará tres veces a la función `randomLED()` que tiene como parámetro el número de milisegundos que tiene que mantener encendido cada led en el momento de hacer “la tirada” del dado, donde se apagan y encienden los leds. Dentro de la función `randomLED()` obtenemos un número aleatorio entre 1 y 6 la función `random()` que devuelve un número aleatorio entre los dos números dados (el segundo no está incluido, por eso ponemos 7). Una vez obtenido ese número, encendemos el led correspondiente a él y lo mantenemos encendido un número determinado de milisegundos dados por el parámetro de la función. En caso especial es cuando el tiempo de espera es 0, que mantiene el led encendido hasta que exista una nueva pulsación; como estamos en pull-up, las lecturas al puerto del botón devolverán `HIGH` hasta que esté pulsado, por lo que para este bucle infinito hasta que haya pulsación podemos usar:

```

do {
// pausamos hasta que haya pulsación

```

```
}while (digitalRead(0));
```

## Salidas analógicas

Las salidas analógicas como tales no existen en Arduino, no podemos decir directamente que una salida tenga un voltaje de 3,2V. Para obtener estas salidas analógicas lo que hacemos es utilizar pulsos de una salida digital, indicando el tiempo que debe estar activo ese pulso. Así, si en la salida HIGH tenemos 5V y le decimos que tiene que estar la mitad del ciclo activo, es como si se tuviera 2,5V aunque realmente no sea así, pero el efecto es el mismo. La llamada tiene la forma `analogWrite(pin, value)`, donde `pin` es el número de pin donde se quiere escribir y `value` es la cantidad del ciclo que debe estar activo, siendo 0 para siempre inactivo (0V) y 255 para siempre activo (5V). Estas salidas solo se pueden obtener en los pines marcados en la tarjeta con la etiqueta **PWM**.

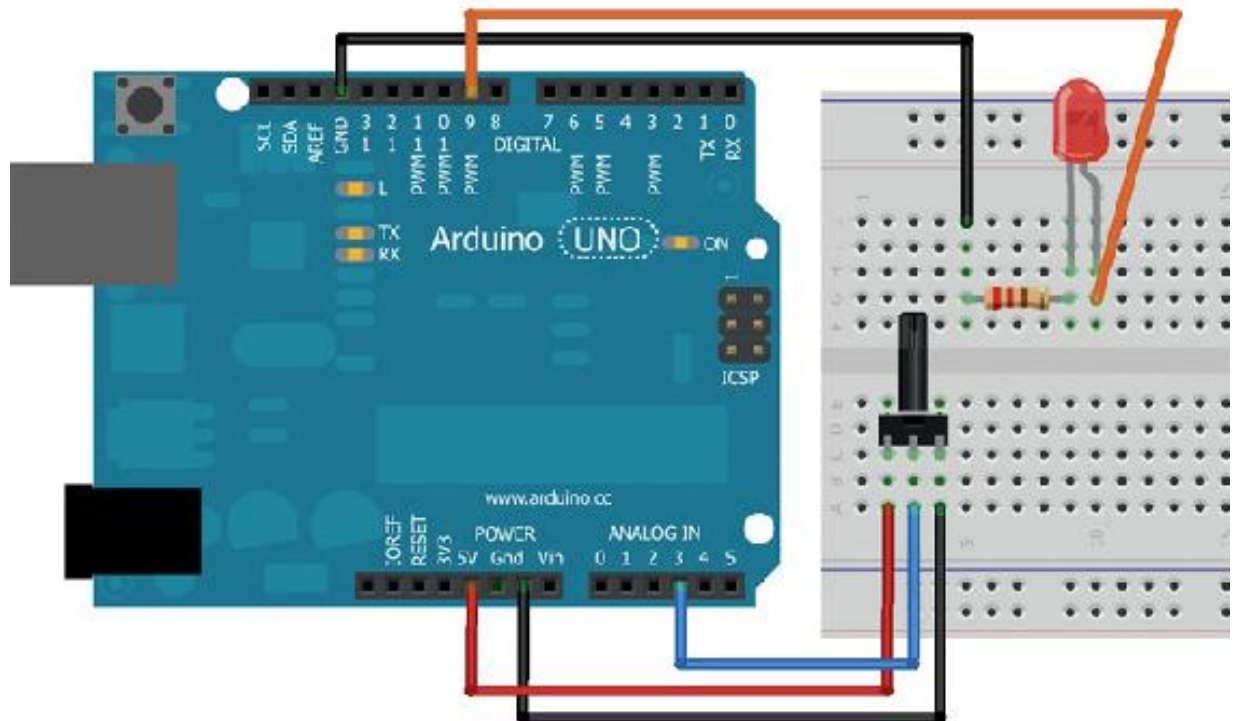


Figura 6.15. PWM con potenciómetro



Esta técnica la usaremos más adelante con los motores, pero por el momento podemos ver la manera en la que afecta a la luminosidad de un led. Vamos a utilizar un circuito semejante al que hemos usado anteriormente, pero esta vez alimentaremos el led con una salida PWM y para variar la pulsación usaremos un potenciómetro. El circuito quedaría como en la figura 6.15.

En esta ocasión hay que tener mucho cuidado de seleccionar bien los pines que utilizamos al hacer las conexiones, ya que para la lectura analógica necesitamos el pin 3 del bloque analógico de la tarjeta y para alimentar al led, tiene que ser una salida que tenga la serigrafía **PWM** o ~, ya que de lo contrario no funcionará; en la tarjeta Arduino UNO, el pin 9 digital tiene esta característica.

El código sería:

```
int ledPin = 9; // LED conectado a pin digital 9
int analogPin = 3; // potenciómetro a pin analógico 3
int val = 0; // variable para el valor leído

void setup(){
  pinMode(ledPin, OUTPUT); // el pin será de salida
}

void loop(){
  val = analogRead(analogPin); // leemos la entrada analógica
  analogWrite(ledPin, val / 4); // analogRead lee de 0 a 1023, analogWrite usa valores de
  0 a 255
}
```

Al ejecutar el sketch y variar el potenciómetro, veremos cómo varía también la intensidad de la luz emitida por el led.

Este sketch no tiene muchas cosas nuevas de lo ya aprendido, se preparan los pines que vamos a utilizar y en el `loop()` nos encargamos de leer el valor dado por la entrada del potenciómetro y la usamos como salida para el led. Hay que tener en cuenta que la lectura nos dará un valor entre 0 y 1023 mientras que la escritura espera que sea un valor entre 0 y 255 por lo que dividimos el valor leído entre 4 antes de entregarlo a la salida.

Si no queremos estar haciendo cuentas matemáticas, existe una función que dado dos rangos de valores, convierte de un rango al otro un valor pasado por parámetro; dicha función es `map()`. Esta función tiene la forma `map(valor, minOrig, maxOrig, minDest, maxDest)`, donde `valor` es el valor a traducir y el resto de parámetros son las escalas: `minOrig` es el mínimo de la escala origen, `maxOrig` el máximo de la escala origen, `minDest` el mínimo de la escala destino y `maxDest` el máximo de la escala destino. Lo que hace la función es transformar el `minOrig` en `minDest`, el `maxOrig` en `maxDest` e interpolar los valores intermedios, devolviendo el valor transformado que le correspondería al valor introducido como parámetro. Lo que hace la función es transformar el `minOrig` en `minDest`, el `maxOrig` en `maxDest` y hacer una especie de regla de tres interpolando los valores intermedios y después devuelve el valor dado como parámetro transformado a la nueva escala. Siempre devuelve valores enteros. En nuestro caso en lugar de dividir entre 4, podríamos haber usado:

```
int newVal = map(val, 0, 1023, 0, 255); //transformación
```

# 7

## Sensores



**En este capítulo aprenderá a:**

- Configurar los sensores
- Usar fotorresistencias
- Utilizar termistores
- Medir temperaturas
- Realizar una estación meteorológica
- Usar librerías

Los sensores se encargarán de tomar datos del mundo externo para ofrecérselos a la tarjeta Arduino a través de alguna de sus entradas; estas entradas pueden ser analógicas o digitales dependiendo de la naturaleza de lo captado y del sensor. Los sensores son muy variados, tanto como las cosas que podemos medir o detectar, por ejemplo temperaturas, distancias, presencias, presiones, cantidad de luz... del mismo modo, podemos encontrarlos de distintas formas:

- Independientes: Son componentes sin soporte alguno, como suelen venir las resistencias, con el elemento y sus terminales de conexión, nada más.
- Montados en placas: Muchos sensores utilizan la técnica de la división de tensión para dar su resultado, es decir, modifican una de las resistencias del divisor de tensión para crear una variación en el voltaje. Para poder utilizar en un circuito estos sensores, deberemos realizar nosotros el montaje mediante una resistencia externa; en lugar de esto, existen placas con el sensor y las resistencias necesarias ya montadas para realizar la medida. Normalmente suelen tener tres terminales, siendo uno de ellos el de tensión, otro el de señal y por último otro con tierra. En ocasiones estas placas incluyen también un potenciómetro para poder calibrar y mejorar la medida.
- Montados en *shields*: Son placas destinadas a utilizar exclusivamente con Arduino (y tarjetas compatibles). Incorporan uno o varios sensores y todos los elementos necesarios para su funcionamiento: leds, resistencias, condensadores... Simplemente hay que ponerlos encima de la placa Arduino para poder utilizarlos.

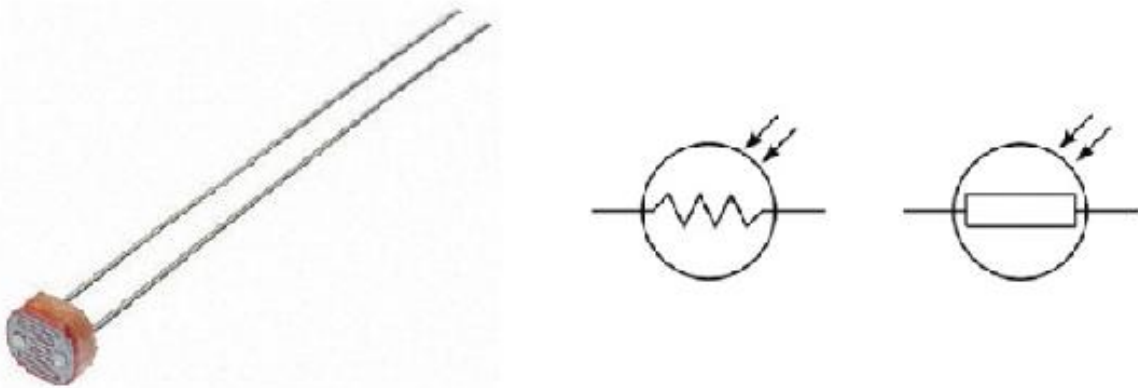
Por ejemplo un detector de luz muy probablemente lo encontremos suelto, pero un detector de posición lo más seguro es que esté en una *shield*, ya que es más complejo. Sean como sean, el funcionamiento es muy similar entre ellos, la medición en el entorno físico real, se traduce a una variación de tensión en el circuito; nosotros simplemente tendremos que interpretar

esta variación. Es muy recomendable leerse las instrucciones del fabricante del sensor, ya que puede que sensores semejantes tengan los pines colocados de manera distinta.

Vamos a ver alguno de estos sensores.

## Fotorresistencia

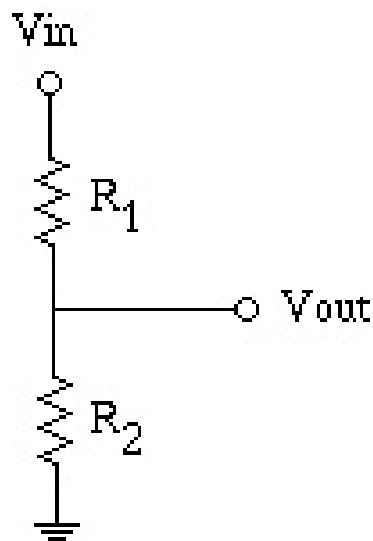
Las fotorresistencias son unos componentes que nos permiten detectar una variación de luminosidad; la manera en la que funcionan es variando su resistencia con la intensidad de luz, concretamente su resistencia disminuye cuanto mayor sea la intensidad de luz recibida. Son conocidas de múltiples formas, tales como fotorresistores, sensores de luz, células fotoeléctricas o con sus siglas en inglés LDR, *light-dependent resistor* (resistencia dependiente de la luz).



**Figura 7.1.** Fotorresistencia

Lo normal es que estos componentes tengan una resistencia del orden de mega ohmios ( $1M\Omega$  o más) cuando están en la oscuridad y desciende hasta unos cientos de ohmios cuando reciben luz. La variación de un valor de resistencia a otro no es inmediata, tarda un tiempo en ajustarse a su nuevo valor, y esto hace que si necesitamos detectar rápidamente si hay o no luz, tengamos usar otros sensores más rápidos (y normalmente más caros).

Cuando usamos fotorresistencias no tenemos que preocuparnos de su polarización, es decir, son como las resistencias y no como los leds, da igual que patilla pongamos con mayor tensión. La manera de trabajar con ellas es mediante el ya conocido divisor de tensión, que podemos ver en la figura 7.2, tomando la medida de la tensión en el punto de unión de la resistencia con la fotorresistencia. Las fotorresistencias se pueden encontrar en placas con el divisor de tensión ya preparado o bien sueltas teniendo que realizar nosotros el montaje de división de tensión.

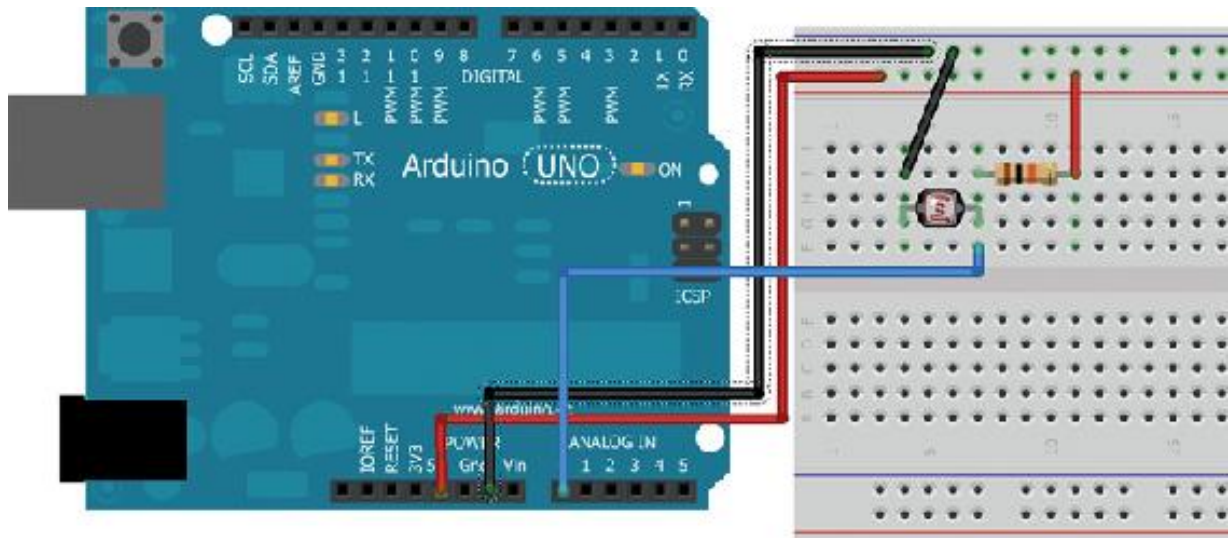


$$V_{out} = \frac{R_2}{R_1 + R_2} V_{in}$$

**Figura 7.2.** Divisor de tensión

Vamos a realizar un pequeño ejemplo donde usaremos una fotorresistencia y mostraremos las lecturas en el monitor serie.

Para el montaje necesitaremos una fotorresistencia y si no está montada en placa, una resistencia externa de 10KΩ para el divisor.



**Figura 7.3.** Circuito con fotorresistencia

En el circuito de la figura 7.3, vemos que tomamos la lectura sobre una entrada analógica (tendremos valores entre 0 y 1023) y realmente estamos midiendo la caída de tensión en la fotorresistencia, de modo que cuando no haya luz, su resistencia interna de éste será de mega ohmios, mucho mayor que los 10K $\Omega$  de la resistencia de referencia; por lo que tendremos en la entrada una lectura cercana a 1023 (5 voltios) y cuando haya luz, la resistencia de la fotorresistencia será de unos cientos de ohmios, que hará que la lectura sea cercana a 0.

El *sketch* simplemente se debe encargar de leer la entrada A0 y mostrarla en el monitor.

```
void setup() {  
  Serial.begin(9600);  
  pinMode(A0, INPUT);  
}  
void loop() {  
  int value = analogRead(A0);  
  Serial.print("Lectura : ");  
  Serial.print(value);  
  Serial.print(" Voltios: ");  
  Serial.println(value * 5.0 / 1023.0);  
  delay(1000);  
}
```

El código no nos tiene que parecer extraño, puesto que hemos realizado semejantes en ocasiones anteriores, pero hay un pequeño detalle. En el cálculo del voltaje vamos a tener decimales, así que tenemos que tener cuidado con no perderlos; si el cálculo se hiciera con `value*5/1023` nos daría el voltaje sin decimales por ser un cálculo entero, por eso marcamos con un `.0` detrás de las constantes y conseguiremos el cálculo en coma flotante y respetar los decimales.

En el monitor serie iremos viendo los valores leídos de la entrada analógica y su transformación a voltios.

```
Lectura : 1003 Voltios: 4.90
Lectura : 974 Voltios: 4.76
Lectura : 943 Voltios: 6.60
Lectura : 531 Voltios: 2.60
Lectura : 560 Voltios: 2.73
Lectura : 372 Voltios: 1,81
Lectura : 222 Voltios: 1.08
```

En el monitor serie iremos viendo los valores leídos de la entrada analógica y su transformación a voltios.

Aunque parezca que el circuito y el programa no sean muy útiles, nos sirven para poner las bases a la hora de realizar por ejemplo una lámpara controlada por la luminosidad de la habitación, de modo que cuando caiga la noche se encienda de modo automático o crear un detector de presencia poniendo una luz apuntando a la fotorresistencia y si algo se interpone entre la luz y el sensor, detectar esa caída de tensión o incluso un detector de humos, donde el humo al estar presente cerca del sensor hará que reciba menos luz.

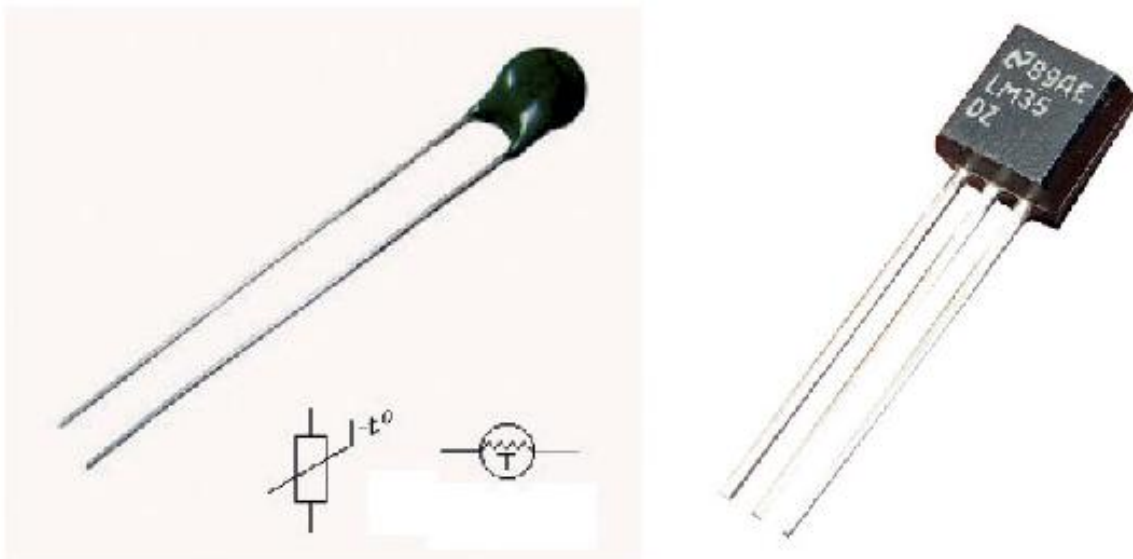
## **Termistores**

Los termistores son unas resistencias que varían su valor dependiendo de la temperatura a la que se encuentren; lo que nos permite nuevamente



aprovecharlas para realizar un cálculo de la caída de tensión en un divisor de tensión, y sabiendo la caída calcular la temperatura.

Una cosa que tenemos que tener clara sobre la medición de la temperatura, es que la escala es totalmente artificial, da igual en qué unidades se mida la temperatura, si grados centígrados o Celsius (los que estamos acostumbrados en Europa), Fahrenheit (usada en EEUU), Kelvin, Reaumur... todas funcionan igual: alguien puso el cero de la escala basándose en un estado físico de algo y luego midió otro estado y se dividió la escala, por ejemplo en la Celsius, el origen de escala es la temperatura a la que se congela el agua. Los grados se calculan tomando la temperatura a la que hierve el agua, restándole la temperatura de congelación y dividiendo este incremento de temperatura en 100 partes; cada una de estas cien partes de incremento obtenidas, se corresponde con un grado Celsius.



**Figura 7.4.** Distintos termistores y su símbolo gráfico

Cuando trabajamos con termistores, éstos normalmente vienen preparados para trabajar en una escala concreta (pero pueden ser utilizados para calcular la temperatura en cualquier escala mediante operaciones matemáticas). Por ejemplo dos termistores muy usados son el LM34 para

Fahrenheit y LM35 para Celsius. De estos termistores hay varios modelos a su vez, cubriendo diferentes rangos de temperaturas ya que un termistor solo puede medir ciertos grados. Está claro por lo que hemos dicho anteriormente que tenemos que tener cuidado en elegir el termistor correcto, ya que si queremos medir la temperatura ambiente, no vamos a usar un termistor con un rango de 100 a 120 grados. Para el ejemplo usaremos el LM35D (o alguno similar) que es muy popular y mide temperaturas de 0 a 100 grados.

Vamos a realizar un ejemplo en el que se mostrará en el monitor serie cada una de las temperaturas leídas por el sensor. El circuito que tenemos que montar con la tarjeta y el termistor es el mostrado en la figura 7.5.

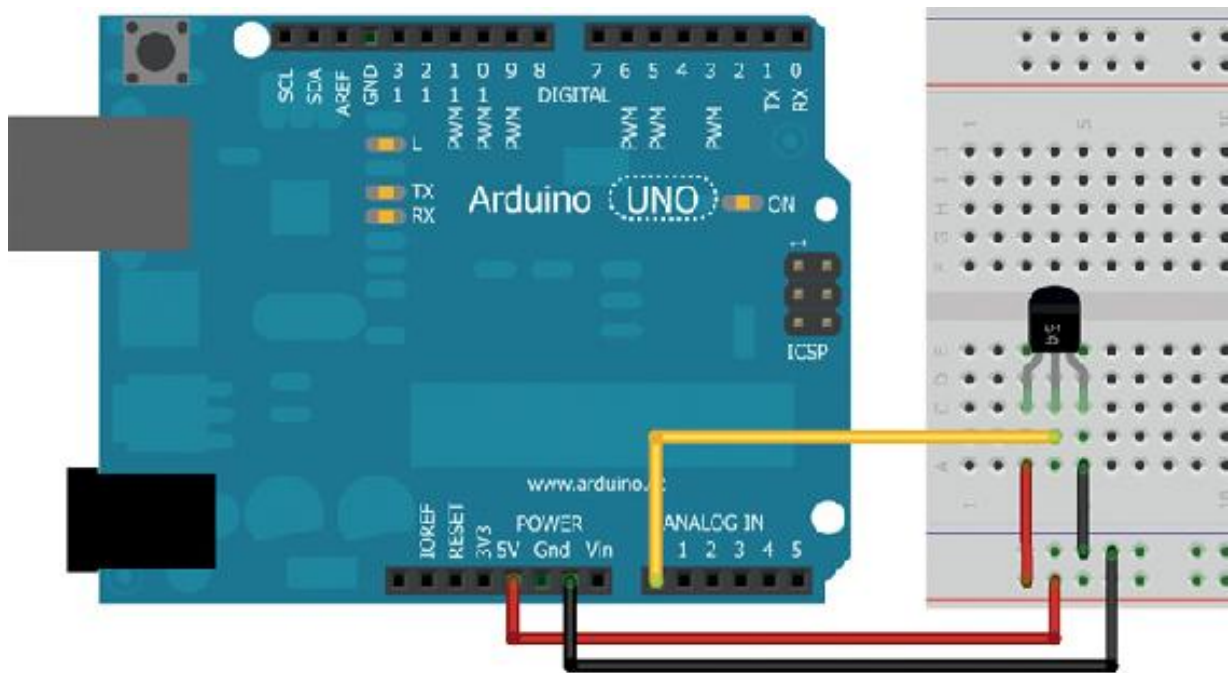


Figura 7.5. Circuito con sensor LM35

En el *sketch* debemos leer la entrada analógica 0 (como muestra el circuito), pero vamos a leer un voltaje y no una temperatura, así que para obtener la temperatura necesitaremos hacer una serie de cálculos que dependen del termistor utilizado, estos datos se obtienen de las hojas de características que nos dan los fabricantes. Vamos a recordar que las entradas analógicas nos van a devolver un valor entre 0 y 1023,

dependiendo de la lectura que se ha tenido, y será 1023 si la lectura es de 5V, ya que es la referencia que utiliza el conversor es de 5V. Para el LM35, la tensión máxima de salida es 1 voltio, eso quiere decir que si tomamos de referencia los 5 voltios que proporciona Arduino, como hemos hecho en ejemplos anteriores, estaremos desperdiciando 4 voltios, ¡que es un 80%!, así que en lugar de eso, usaremos una referencia interna que está disponible en Arduino, que es de 1.1 voltios. Sabiendo que podemos leer un rango de valores de voltaje entre 0 y 1.1V y 1024 valores posibles (de 0 a 1023), el cálculo es  $1.1V/1024 \text{ valores} = 0.00107421875V = 1.074mV$ , es decir podemos hacer incrementos de lectura de temperatura de 1.074mV; si hiciéramos el mismo cálculo con los 5 voltios tendríamos unos incrementos mayores, o lo que es lo mismo, más error en la lectura de la temperatura. En las hojas del fabricante de este componente nos dice que puede leer de 0 a 100 grados de manera lineal (es decir, que la tensión medida varía del mismo modo que la temperatura) con un factor de +10mV/°C (10 milivoltios por cada grado centígrado), o dicho de otro modo, que si hace 1°C, tendríamos una tensión de 10mV. Además antes hemos calculado que cada valor representado por la lectura analógica en este caso son 1.074mV, eso quiere decir que para representar 1°C necesitaremos  $10mV/1.074mV = 9.31$ , o dicho de otra manera, si hace 1°C, Arduino nos devolverá un valor de 9 o 10 en la lectura analógica. Este 9.31 es un número mágico para nuestro *sketch* porque nos servirá para transformar entre la lectura de la entrada analógica y el valor de temperatura a mostrar. El *sketch* sería:

```
float tempC; //temperatura en celsius
int analogValue; //valor leído en el puerto
int sensorPin = A0;

void setup() {
  analogReference(INTERNAL);
  Serial.begin(9600);
}

void loop() {
  analogValue = analogRead(sensorPin);
  tempC = analogValue / 9.31; //se divide el valor leído por el número calculado
  Serial.print("El valor obtenido son ");
  Serial.print(tempC);
```

```
Serial.println(" grados Celsius");  
delay(1000);  
}
```

Para la obtención de la temperatura simplemente hemos dividido el valor leído en la entrada por el número de valores que se necesitaba para representar cada grado centígrado, que habíamos calculado que era 9.31. Si ejecutamos el programa, ya podemos ver la temperatura en nuestro monitor serie.

## **Sensor de humedad**

Si quisiéramos hacer una estación meteorológica para casa, necesitaríamos también un sensor que nos dijera la humedad que tenemos en la habitación. Aunque podemos encontrar sensores de humedad por separado, normalmente se encuentran acompañados de otros sensores de temperatura, presión o ambos. Sensores muy comunes y baratos de este tipo son los DHT11, DHT12, DHT21... que se diferencian en que el DHT11 es un poco peor (pero más barato) y además de medir la humedad son capaces de medir la temperatura, con lo que podemos hacer una pequeña estación meteorológica en casa; vamos a ello.

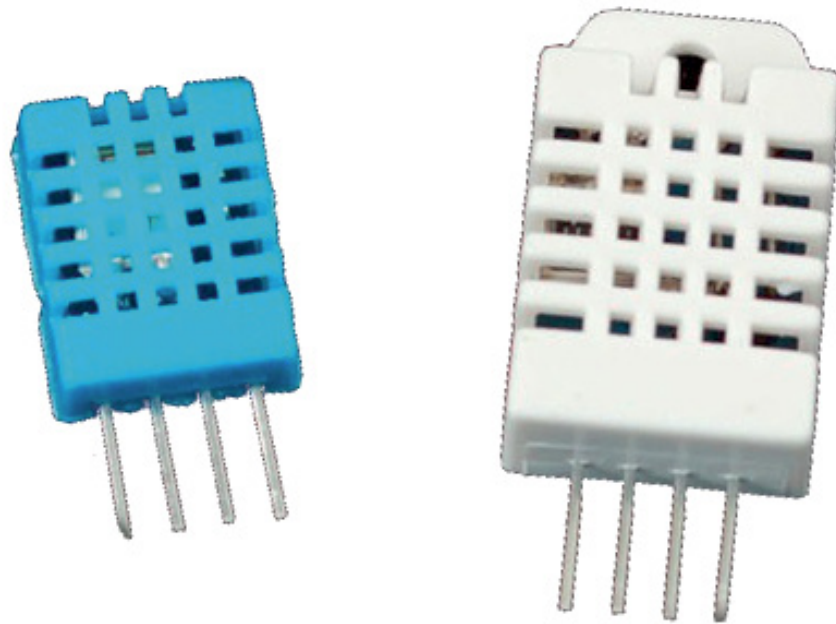


Figura 7.6. Sensores DHT21 y DHT22

El montaje para su uso es muy sencillo como podemos ver en la figura 7.8, pero el *sketch* se nos complicará un poco. El componente tiene cuatro pines (también existen versiones de solo tres pines), de los cuales usaremos tres y dejaremos el cuarto libre. Como hemos hecho anteriormente, alimentaremos el sensor conectando un pin a los 5V de la placa, otro a tierra y el último lo usaremos para obtener la medida, que en nuestro caso será leído por el pin digital 2. El problema es que se usa el mismo pin para la medida de temperatura y la de humedad... y es más, usamos un pin digital cuando la temperatura y la humedad son valores analógicos; el fabricante tendrá que explicarnos cómo funciona ya que nos está dando la información codificada.

Si miramos las hojas del fabricante sobre este sensor (podemos buscarlas en Internet, también nos sirven las de algún sensor compatible por ejemplo <http://www.electrodragon.com/w/images/6/6f/DHT21.pdf>), nos dirán la manera en la que debemos interpretar lo que se lee por el pin de entrada.

En una transmisión codificada, tanto el emisor (el sensor) como el receptor (la tarjeta Arduino) tienen que saber qué información se está transmitiendo y cómo se está haciendo, y tienen que ir perfectamente acompasados, no sea que por ejemplo el sensor esté dando la temperatura y Arduino se piense que está leyendo la humedad. Para ir perfectamente alineados, la transmisión se comienza con lo que se denomina *handshake* (choque de manos); el sistema receptor (nuestra tarjeta Arduino) enviará una serie de 1 y 0 con una duración y orden concretos (dados por el fabricante) y a partir de ese momento el sensor sabe que tiene que enviar los datos. Los datos que se nos enviarán, también vienen codificados en 40 bits (según el fabricante) y aunque no sea necesario saberlo, para los más inquietos, la transmisión se realiza de la siguiente manera: se envía durante  $50\mu\text{s}$  el estado *LOW* indicando el comienzo de un bit a transmitir y si los siguientes  $28\mu\text{s}$  son en *HIGH* indica que se envía un 0 pero si está en *HIGH*  $70\mu\text{s}$  significa que lo que se transmite es un 1; para la transmisión del siguiente bit vuelve a tener una salida *LOW*. Los 40 bits que se transmiten son 5 bytes (grupos de 8 bits) que se distribuyen de la siguiente manera, primer byte se refiere a la parte entera de la humedad, el segundo byte representa la parte decimal de la humedad, el tercer byte es la parte entera de la temperatura, el cuarto byte es la parte decimal de la temperatura y por último el quinto byte sirve para comprobar que los datos han sido correctamente recibidos y guarda una suma de los cuatro bytes anteriores; si la suma de los cuatro bytes no es igual al quinto byte, quiere decir que hemos tenido un problema durante la transmisión y los datos leídos no son válidos.

Si no se ha comprendido el funcionamiento de la transmisión, no hay que preocuparse, es bastante complicado así que usaremos una librería que nos ayude a comprender el valor leído en cada momento. Una librería es una colección de código que “alguien” ha escrito para facilitar tareas, en nuestro caso ayudarnos a utilizar el sensor DHT21. Para utilizar una librería simplemente tenemos que añadir al principio del *sketch* la línea:

```
#include "nombre_de_la_libreria"
```

La librería que usaremos es una específica para los sensores DHT, y podemos encontrarla en la web GitHub (<https://github.com/adafruit/DHT-sensor-library/archive/master.zip>). Para que el editor de Arduino pueda usar la librería, tenemos que colocarla en algún lugar donde lo pueda encontrar. Tras descargar el archivo, lo descomprimos y renombramos el directorio obtenido a DHT y lo copiamos dentro del directorio donde tenemos el programa Arduino IDE, es decir donde instalamos el editor de Arduino, concretamente dentro del directorio `<directorioArduino>\libraries`, donde se encuentran todas las librerías disponibles. Tras copiar la librería, tenemos que reiniciar el editor Arduino para que pueda ser utilizada. Si todo ha ido de manera correcta, tendremos disponible la librería en el menú Sketch > Incluir librerías, tal y como muestra la figura 7.7.



**Figura 7.7.** Menú de librerías

Vamos a ponernos ya manos a la obra con el sensor. Lo primero es realizar el circuito tal y como muestra la figura 7.8. Usaremos un DHT21 y junto a él, una resistencia de 10k $\Omega$ ; si se quiere mayor estabilidad se puede añadir un pequeño condensador de unos 100nF entre la alimentación y tierra, pero es optativo.



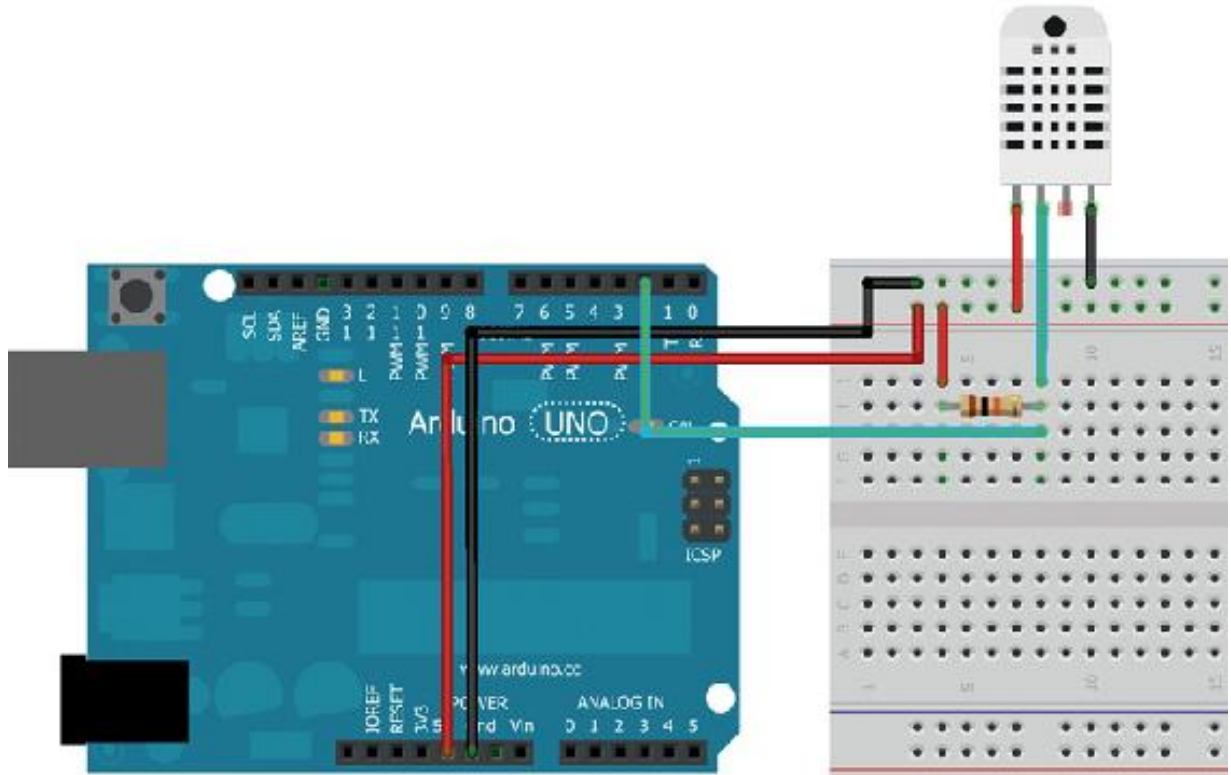


Figura 7.8. Circuito con DHT21

El *sketch* quedaría:

```
#include "DHT.h"
#define DHTPIN 2
#define DHTTYPE DHT21 // DHT 21 (AM2301)

DHT dht(DHTPIN, DHTTYPE);

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

void loop() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();

  if (isnan(t) || isnan(h)) {
    Serial.println("Error en la lectura del sensor");
  } else {
    Serial.print("Humedad relativa: ");
    Serial.print(h);
    Serial.print(" %\t");
    Serial.print("Temperatura: ");
```

```
Serial.print(t);  
Serial.println(" grados Celsius.");  
}  
}
```

Como ya habíamos dicho, lo primero que vemos en el *sketch* es el uso de la librería mediante:

```
#include "DHT.h"
```

con este supuesto debemos decir al programa que tiene que utilizar la librería DHT (la tenemos que tener accesible en nuestro *sketch*, para ello hay que leerse el fichero que viene para su instalación o seguir los pasos descritos en el libro). La línea:

```
DHT dht(DHTPIN, DHTTYPE);
```

indica que la variable *dht* será de tipo *DHT* y se configura dándole como parámetros el pin de datos y el tipo de sensor, que en nuestro caso un DHT21.

En la función `setup()` preparamos el monitor serie para mostrar la información y dentro de la función `loop()` será donde realizaremos las lecturas y mostraremos el resultado obtenido. Las lecturas de humedad y temperatura se hacen mediante:

```
float h = dht.readHumidity();  
float t = dht.readTemperature();
```

y guardamos su valor en unas variables que utilizaremos para mostrarlo en el monitor serie. Tenemos un bloque donde se comprueba que el valor obtenido sea realmente un número, y esto lo hacemos con la función `isnan()` que significa *is not a number* (no es un número). A esta función se le pasa como parámetro una variable que supuestamente tenga un número y la función devuelve un 1 si lo que se le ha pasado NO es un número y un 0 si SÍ que lo es. Si la lectura del sensor falla, en lugar de un número, en las variables *h* y *t* tendremos el valor constante *NAN* (*Not A Number*, no es un número).

Ya podríamos usar nuestra estación meteorológica y ver las lecturas en el monitor serie del entorno Arduino; tenemos que decir que no es muy útil tener que utilizar el monitor serie para ver qué temperatura hace, pero no hay que preocuparse, más adelante sabremos como mostrar estos valores por ejemplo en una página web y poder consultar la temperatura de una habitación de nuestra casa incluso si estamos de vacaciones.

## Sensor de humedad sin librería

Este punto es para aquellos que quieran conocer con más detalle el funcionamiento del sensor, pero no es necesario leerlo si no se quiere, ya que es un poco avanzado. Lo que vamos a hacer es intentar leer e interpretar los valores del sensor sin utilizar las librerías, es decir tenemos que hacer a mano todo el trabajo que la librería hacía por nosotros.

De lo que hemos hablado de cómo funcionaba el sensor, lo que tenemos que hacer es enviar desde la tarjeta Arduino la secuencia de comienzo de transmisión y escuchar el puerto de entrada detectando la duración de las lecturas *HIGH* para saber si se trata de un 0 o un 1; una vez obtenidos los 5 bytes, se comprobará que el quinto byte coincide con la suma de los otros cuatro y se mostrará el resultado en pantalla convenientemente formateado.

Para leer el sensor utilizaremos el pin 2, ya que al ser una transmisión en serie no necesitamos que sea un puerto analógico (se podría usar) y usaremos una variable global para retener los 5 bytes de cada lectura y otro byte para mantener errores que se puedan producir durante la misma. Comenzamos el *sketch* con estas variables.

```
const byte dhtPin = 2;
byte errorCode; // código de error
byte dhtData[5]; //bytes leídos
```

En la función `setup()` configuraremos el pin 2 como de salida y en `HIGH`, ya que aunque queremos leer, aún no hay datos que leer y se tiene que enviar la secuencia de activación de la lectura.

En la función `loop()` leeremos los 40 bits y mostraremos el resultado de la lectura que puede ser satisfactoria o no dependiendo de la variable `errorCode` que tendremos que rellenar más adelante. Los errores que se pueden producir son: que no se haya enviado o recibido el *handshake* de comienzo de transmisión, que el byte de chequeo no corresponda con la suma de los otros bytes o errores desconocidos.

```
void loop(){
  readDHT(); // comenzamos la lectura
  switch (errorCode){
    // dependiendo del código de error mostrar en pantalla el resultado
    // o mostrar una descripción del error
  }
}
```

La función `readDHT()` que se llama desde el bucle principal debe obtener del sensor los 5 bytes con información, pero antes se debe enviar el *handshake*, que según el fabricante debe ser mínimo 500ms en *HIGH* para pasar a *LOW* durante 20 o 40µs:

```
digitalWrite(dhtPin,LOW); //handshake de comienzo
delay(20);
digitalWrite(dhtPin,HIGH);
delayMicroseconds(40);
```

Una vez enviado el *handshake* se espera a que el sensor devuelva el suyo, que debe ser también un *LOW* durante 80µs para pasar a *HIGH* durante otros 80µs por lo que se debe poner el pin en modo lectura, leer esperando recibir un *LOW*, esperar 80µs, volver a leer esperando un *HIGH* y si todo ha ido bien las siguientes lecturas ya corresponderían a datos; en caso de no darse bien (que no leamos el dato que se espera), se guarda un código de error en la variable *errorCode*.

```
dhtIn=digitalRead(dhtPin); // Lectura posible error
//Condición 1 de comienzo
if(dhtIn){// si se lee HIGH => error
  errorCode=1;
  return;
}
```

```
delayMicroseconds(80);// espera transición LOW HIGH
```

```
dhtIn=digitalRead(dhtPin); // Lectura posible error
//Condición 2 de comienzo
if(!dhtIn){// si se lee LOW => error
errorCode=2;
return;
}
```

Una vez se comienzan a recibir los datos, se deben almacenar en cada uno de los bytes del *array* para luego mostrarlos en pantalla, así que los rellenamos mediante un bucle y con ayuda de la función `readDHTByte()` que se encargará de interpretar los datos recibidos por el puerto. Un *array* es una variable que puede guardar muchos valores de un mismo tipo. Para acceder a los distintos valores solo tenemos que escribir el nombre de la variable y entre corchetes el número de elemento al que queremos acceder; además de un número, podemos poner una variable que contenga un número, por ejemplo para este caso usaremos un bucle donde *i* va incrementando el valor, y así recorreremos el *array* de 5 elementos:

```
for (i=0; i<5; i++){
dhtData[i] = readDHTByte();
}
```

Una vez recibidos todos los bytes se debe comprobar que la suma de los cuatro primeros es igual al quinto, de lo contrario se generará un error.

```
byte dht_check_sum =
dhtData[0]+dhtData[1]+dhtData[2]+dhtData[3];
if(dhtData[4]!= dht_check_sum){
errorCode=3;
}
```

Vamos a entrar ahora a ver la función `readDHTByte()`, que como dijimos se encargará de interpretar los datos que se vayan leyendo en el terminal. Los datos se transmiten enviando un *LOW* durante  $50\mu\text{s}$  y luego se pasa a *HIGH*, si la duración de *HIGH* es de entre  $26$  y  $28\mu\text{s}$  quiere decir que es un 0, si es de  $70\mu\text{s}$  quiere decir que es un 1; lo que haremos será por cada bit, leer mientras esté a *LOW*, cuando pase a *HIGH* esperar  $30\mu\text{s}$  y si la lectura es nuevamente *HIGH* quiere decir que es un 1 (ya que hemos pasado los  $28\mu\text{s}$ ) y si no será un 0; una vez obtenido el valor, lo

almacenamos y volvemos a esperar leyendo hasta que deje de ser *HIGH* que significará que comienza un nuevo bit.

```
for(i=0; i< 8; i++){
while(digitalRead(dhtPin)==LOW);
delayMicroseconds(30);
if (digitalRead(dhtPin)==HIGH){
result |= (1<<(7-i));
}
while (digitalRead(dhtPin)==HIGH);
}
```

Para no alargar esta función, hemos usado unas sentencias abreviadas que vamos a explicar. La primera es:

```
while (digitalRead(dhtPin)==LOW);
```

Con esta sentencia le decimos que se quede leyendo mientras sea la entrada *LOW*, realmente es como si hubiéramos escrito:

```
while(digitalRead(dhtPin)==LOW){
}
```

es decir, que mientras sea *LOW* no haga nada. Esta misma técnica se ha usado para las lecturas *HIGH*. La segunda sentencia abreviada es:

```
result |= (1<<(7-i));
```

que realiza un desplazamiento del 1 tantas posiciones como corresponda dependiendo del valor de la variable *i* y efectúa un “or” lógico del valor obtenido en el desplazamiento con el valor guardado en la variable *result*, guardando este nuevo resultado otra vez en *result*, o dicho de otro modo, pone un 1 en el byte representado por *result*, en la posición correspondiente dependiendo de la variable *i*. Teniendo en cuenta lo explicado anteriormente y el código completo del *sketch* quedaría:

```
const byte dhtPin = 2;
byte errorCode; // código de error
byte dhtData[5]; //bytes leídos
void setup(){
```

```

pinMode(dhtPin,OUTPUT);
digitalWrite(dhtPin,HIGH);
Serial.begin(9600);
}

void loop(){
readDHT(); // comenzamos la lectura

switch (errorCode){

case 0: // no hay errores mostramos la salida
Serial.print("Humedad relativa: ");
Serial.print(dhtData[0], DEC);
Serial.print(".");
Serial.print(dhtData[1], DEC);
Serial.print(" %\t");

Serial.print("Temperatura: ");
Serial.print(dhtData[2], DEC);
Serial.print(".");
Serial.print(dhtData[3], DEC);
Serial.println(" grados Celsius.");

break;
//Errores
case 1:
Serial.println("Error 1: Ha fallado la primera condición de comienzo.");
break;

case 2:
Serial.println("Error 2: Ha fallado la segunda condición de comienzo");
break;

case 3:
Serial.println("Error 3: Error de checksum.");
break;

default:
Serial.println("Error no conocido.");
break;
}

delay(1000); // un segundo entre lecturas
}

/*****
* Inicia la secuencia de envío de datos y recibe los
* 40 bits de información con temperatura y humedad
* Se encarga de generar los tres códigos posibles de error

```

```

*****/
void readDHT(){
errorCode=0;//limpiamos último error
    byte dhtIn;
    byte i;

    digitalWrite(dhtPin,LOW); //handshake de comienzo
    delay(20);
digitalWrite(dhtPin,HIGH);
delayMicroseconds(40);
pinMode(dhtPin,INPUT);

dhtIn=digitalRead(dhtPin); // Lectura posible error
//Condición 1 de comienzo
if(dhtIn){// si se lee HIGH => error
errorCode=1;
return;
}

delayMicroseconds(80);// espera transición LOW HIGH

dhtIn=digitalRead(dhtPin); // Lectura posible error
//Condición 2 de comienzo
if(!dhtIn){// si se lee LOW => error
errorCode=2;
return;
}

delayMicroseconds(80);

//lectura byte a byte
for (i=0; i<5; i++){
dhtData[i] = readDHTByte();
}

// se pasa a high para la siguiente lectura
pinMode(dhtPin,OUTPUT);
digitalWrite(dhtPin,HIGH);

//comprobación de checksum
byte dht_check_sum =
dhtData[0]+dhtData[1]+dhtData[2]+dhtData[3];
if(dhtData[4] != dht_check_sum){
errorCode=3;
}

};

/*****

```



```

* Lee un byte teniendo en cuenta los tiempos de HIGH
*****/
byte readDHTByte(){
byte i = 0;
byte result=0;

for(i=0; i< 8; i++){
while(digitalRead(dhtPin)==LOW);
delayMicroseconds(30);
if (digitalRead(dhtPin)==HIGH){
result |= (1<<(7-i));
}
while (digitalRead(dhtPin)==HIGH);
}

return result;
}

```

¿Verdad que es más fácil usar librerías?

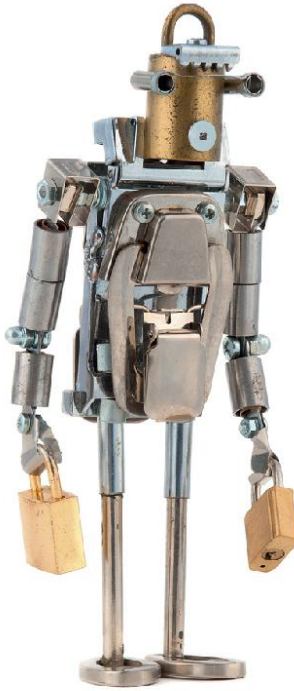
Si queremos comprender cómo funcionan los errores, podemos desconectar el cable del pin 2 y nunca se llevaría a cabo el *handshake*, lo que generaría un error.

## Otros sensores

Existen otros muchos sensores a parte de los que hemos visto, tantos como cosas que se nos ocurran que podamos medir o detectar, por ejemplo hay para medir presiones, para medir campos magnéticos, para medir la posición, etc. Aunque no hayamos visto todos en este capítulo, con lo que hemos aprendido en él, podemos ser capaces de usarlos con muy poco esfuerzo.

# 8

## Salida de audio



**En este capítulo aprenderá a:**

- Conocer los tipos de altavoz más usados
- Usar un altavoz piezoeléctrico
- Realizar una alarma
- Emitir sonidos en Arduino
- Crear un piano

Ya hemos visto cómo Arduino puede enviar al mundo físico unos impulsos eléctricos para mostrar valores por ejemplo encendiendo una luz, pero podemos utilizar también un elemento que transforme esos impulsos eléctricos en sonido. Podemos decir que una onda sonora (lo que oímos) no es más que una presión de aire, con lo que el altavoz lo que hará es precisamente transformar la electricidad en ondas que se desplazarán por el aire.

Existen distintos tipos de altavoces, aunque los más comunes quizá sean los dinámicos, los activos y los piezoeléctricos que son los que vamos a ver.

## **Altavoz piezoeléctrico como sensor**

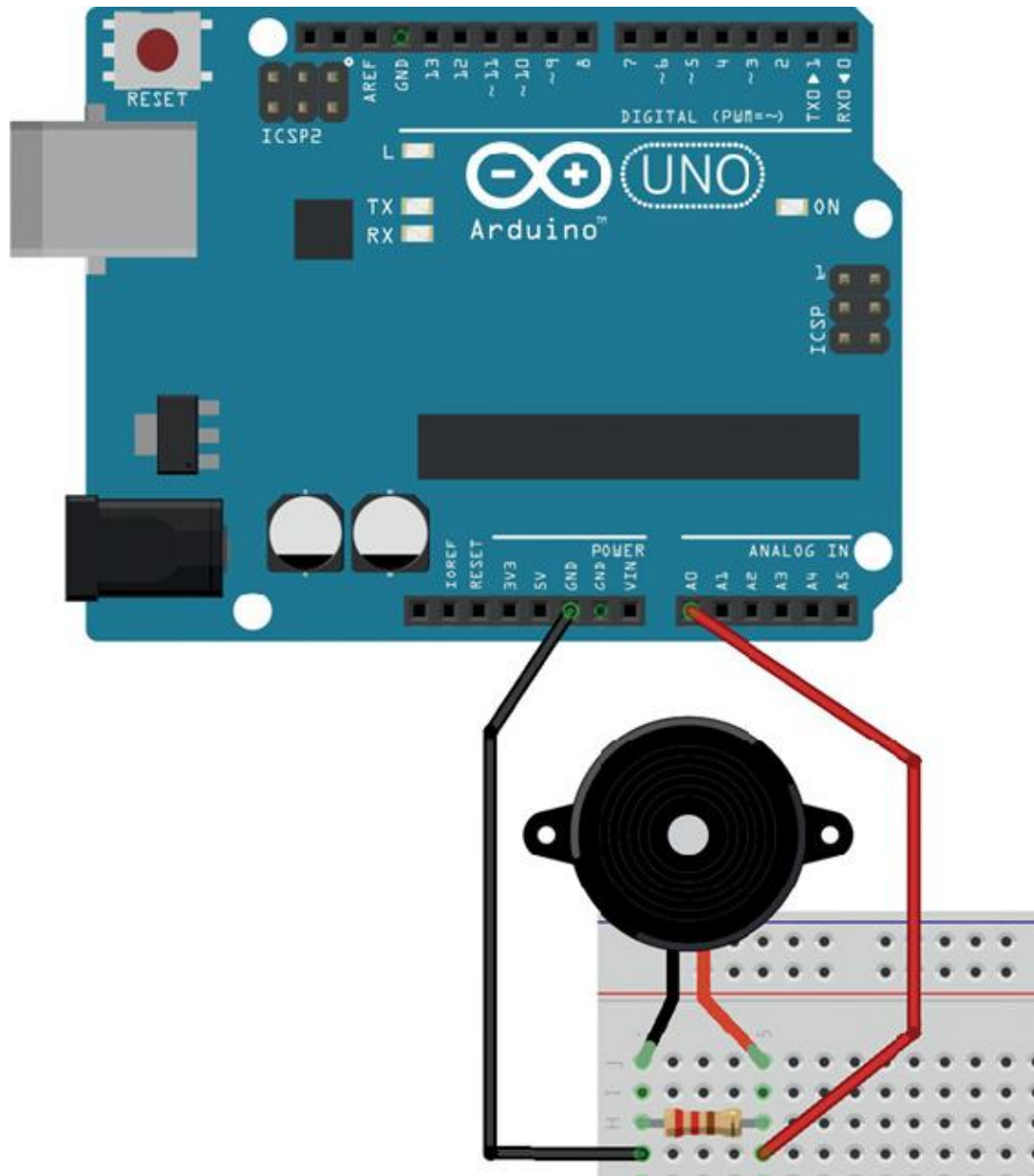
La piezoelectricidad es una propiedad de algunos materiales de verse deformados cuando se les aplica una tensión o diferencia de potencial. Para generar sonido una de sus caras se une a un cono de ampliación que será desplazado generando presión en el aire que generará las ondas sonoras emitiendo pequeños “clics” como si fuera un tambor. Si se cambia la frecuencia de los pulsos eléctricos enviados variará el número de “clics” por segundo conseguidos y así crearemos distintos sonidos. Son fácilmente integrables y especialmente buenos en ultrasonidos. Los podemos encontrar en las tarjetas de navidad o cumpleaños con sonido o en timbres de casa.

Además de deformarse por la diferencia de potencial, sucede al contrario... si lo deformamos, aparecen campos eléctricos en ellos, lo que nos puede servir de detector de impactos, de control de peso e incluso de torsión. Vamos a ver un ejemplo rápido.



**Figura 8.1.** Altavoz piezoeléctrico

Para el circuito, simplemente necesitamos un altavoz piezoeléctrico que montaremos según la figura 8.2.



**Figura 8.2.** Circuito altavoz piezoeléctrico como sensor

En el sketch leeremos la entrada analógica a la que está conectado el altavoz e iremos mostrando los valores leídos en el monitor serie.

```
const int sensorPin = A0; // Altavoz en A0
int value = 0;

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

void loop() {
```

```
// leemos el sensor y lo mostramos en el monitor
  value = analogRead(sensorPin);
  Serial.print("Valor en la entrada: ");
  Serial.println(value);

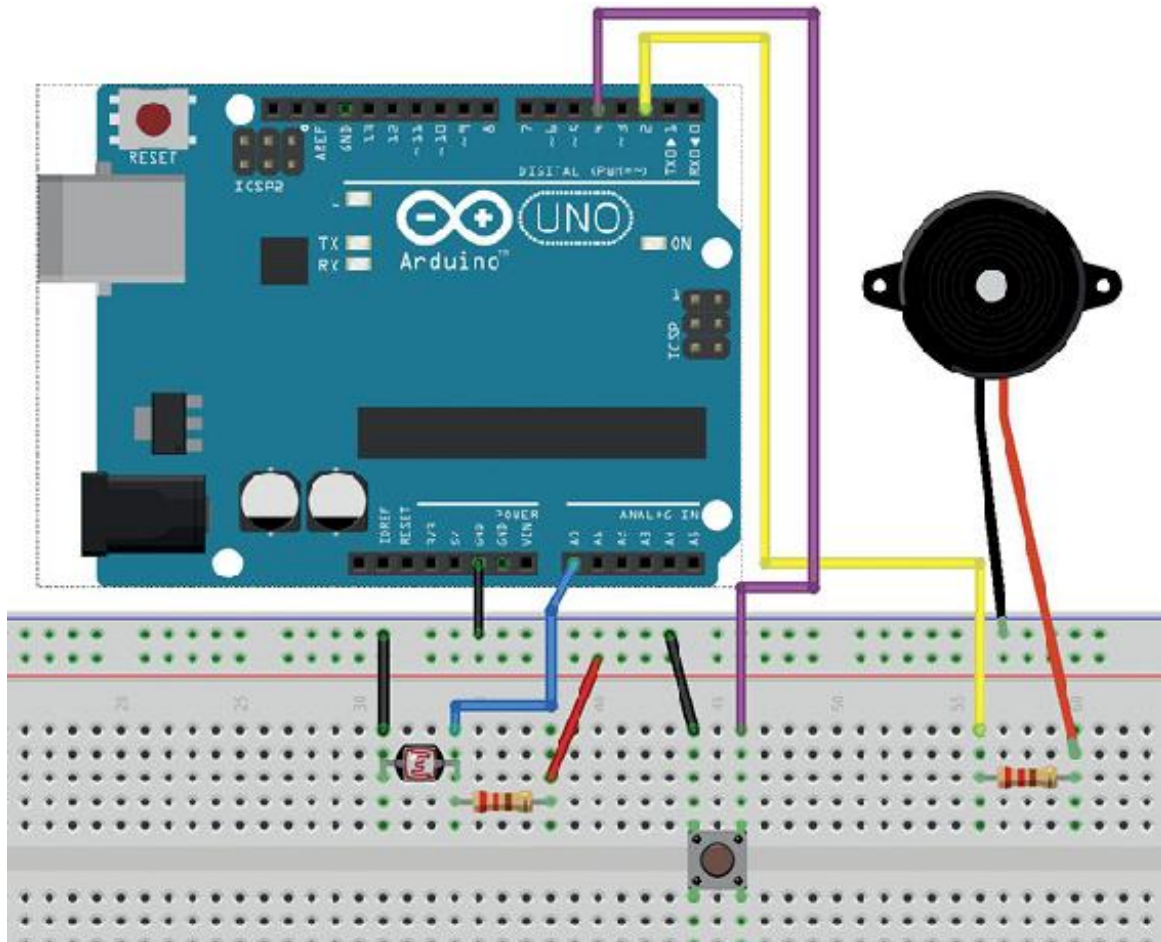
delay(100); // retardo
}
```

Si ejecutamos el programa y vamos golpeando el altavoz, veremos que los valores mostrados en el monitor van variando, con lo que podemos poner un valor límite a partir del cual realizar alguna acción, por ejemplo si detectamos un golpe en el sensor, abrir una puerta.

Aunque hemos visto que el altavoz puede actuar como sensor, no es lo normal. Vamos a ver ahora su funcionamiento como altavoz.

## **Alarma**

Algunos modelos de alarma del mercado, traen consigo un detector de movimientos mediante luz infrarroja, que funciona emitiendo este tipo de luz y recibéndola a la misma vez, de modo que cuando existe una variación en lo recibido, es porque la luz ha “chocado” contra algo y esto significa que hay alguien ahí y la alarma se debe disparar. En este apartado haremos algo parecido, aunque que como no hemos visto cómo utilizar infrarrojos, lo haremos con un LDR. En este caso también detectaremos la presencia mediante la variación de la luz que recoge la fotorresistencia. No es la detección más fiable del mundo, ya que además recordemos que el LDR tarda un poco en variar su valor ante un cambio de luz, pero nos servirá para ver el funcionamiento.



**Figura 8.3.** Circuito para alarma

Una manera de mejorar el rendimiento de la detección es usar un láser apuntando directamente al LDR y la zona “vigilada” será la que recorra la luz del láser, de modo que si alguien o algo se cruza entre el emisor láser y el LDR, creará una sombra y hará que salte la alarma.

En el sketch, lo que haremos será leer la entrada analógica sobre la que se encuentra el LDR y en cuanto pase un umbral establecido, dispararemos la alarma que sonará hasta que pulsemos el botón de desconexión.

Entonces para el circuito necesitaremos un LDR que hará de sensor, una resistencia de  $10K\Omega$  para el divisor de tensión en caso de que no esté ya integrado en el LDR (recordemos que si tiene 2 patillas no tiene divisor de tensión y lo tendremos que hacer nosotros, si tiene 3 sí lo tiene), una luz (si se tiene un diodo laser, mejor, ya que se puede apuntar directamente al

LDR), un botón para controlar la alarma, un altavoz piezoeléctrico y una resistencia para poner en serie con el altavoz y así disminuir el volumen del sonido cuando salte la alarma; el valor de la resistencia puede ser el que se quiera sabiendo que cuanto mayor sea, más bajo será el volumen del sonido, por ejemplo 220  $\Omega$  (o usar un potenciómetro, con el que regular el volumen).

El funcionamiento del sketch será el siguiente. Vamos a considerar que la alarma puede estar en tres estados:

- Apagada: estado 0
- Encendida, armada, vigilando: estado 1
- Disparada, ha saltado la alarma: estado 2

Al comenzar, la alarma se encontrará parada, estará en su estado 0. Una vez pulsamos el botón, estará encendida y vigilando, está en el estado 2, es decir estaremos mirando los valores del sensor por si se produce variación o no en la entrada y también miraremos si se pulsa el botón para pararla. En caso de que salte la alarma, tenemos que pasar al estado 3, donde tiene que sonar la alarma y tenemos que mirar si pulsa el botón de parar (en este caso ya no nos interesa leer el sensor... ya ha entrado el ladrón). Como buena alarma, usaremos un led para mostrar si está encendida o apagada (no sea que nos salte la alarma a nosotros), usaremos el led integrado de la tarjeta Arduino.

Para facilitar el sketch, vamos a utilizar también funciones:

```
const int sensorThreshold = 500; // umbral de la alarma
const int sensorPin = A0; // LDR
const int speakerPin = 2; // altavoz
const int stopPin = 4; // botón de control
const int ledPin = 13; // luz de estado
int alarmStatus = 0; // estado de la alarma

void setup() {
  pinMode(speakerPin, OUTPUT);
  pinMode(stopPin, INPUT);
  digitalWrite(stopPin, HIGH); // se activa la resistencia de pull-up
  pinMode(ledPin, OUTPUT);
}
```



```

void loop() {
// leemos a ver si han pulsado el botón
  if (digitalRead(stopPin) == LOW) {
    //se ha pulsado el botón
    if (alarmStatus == 0) {
      alarmStatus = 1;
    }
    else if (alarmStatus == 1 || alarmStatus == 2) {
      alarmStatus = 0;
    }
    delay(500); //evitar leer más de una vez
  }

  //dependiendo del estado de la alarma sonara, mirará el sensor o no hará nada
  if (alarmStatus == 0) {
    //no se hace nada... está parado
  }
  else if (alarmStatus == 1) {
    //leemos el valor medido
    int sensorValue = analogRead(sensorPin);
    if (sensorValue < sensorThreshold) {
      alarmStatus = 2;//salta la alarma
    }
  }
  else {
    suenaAlarma();
  }
  muestraStatus(alarmStatus);
}

/**
 * luz mostrando el status, encendida o sonando= led encendido
 */
void muestraStatus(int status) {
  switch (status) {
  case 0:
    digitalWrite(ledPin, LOW);
    break;
  case 1:
  case 2:
    digitalWrite(ledPin, HIGH);
    break;
  }
}

/**
 * Sonido de alarma
 */

```

```
void suenaAlarma() {  
  digitalWrite(speakerPin, HIGH);  
  delay(200);  
  digitalWrite(speakerPin, LOW);  
  delay(200);  
}
```

Lo primero que hemos hecho es configurar los pines en la función `setup()`, indicando los que son de entrada, los que son de salida y activando la resistencia de pull-up para el pin del botón.

En el `loop()`, leemos el estado del botón para saber si tenemos que pasar de estado de la alarma; con el botón podemos pasar de parada a encendida (de estado 0 a 1) o de encendida o sonando a parada (de los estados 1 y 2 a 0).

Luego nos encontramos un `if` que nos servirá para saber que hacer dependiendo del estado en el que está la alarma; el estado 0 no tiene que hacer nada puesto que está parada, en el estado 1 tiene que leer el sensor para ver si está su valor por debajo del umbral (quiere decir que hay una sombra, es decir hay alguien) y en tal caso pasar al estado 2, y por último en el estado 2 tiene que hacer sonar la alarma que se hace con la función `suenaAlarma()` que está un poco más adelante.

Por último con la función `muestraStatus(alarmStatus)` indicaremos al led de la tarjeta si tiene que estar encendido o no dependiendo del estado de la alarma, es por eso que se le pasa el parámetro `alarmStatus`.

La función `muestraStatus(alarmStatus)` la definimos tras el bucle principal `loop()`, y tiene la forma:

```
void muestraStatus(int status) {
```

Con ello indicamos que cuando se quiera llamar al código de esta función hay que usar la palabra `muestraStatus`, que le tenemos que informar un parámetro de tipo entero (`int`) que dentro del bloque de la función podremos acceder a su valor mediante la variable `status` (ya que así la hemos llamado en la definición) y que tras acabar la función no devolverá

nada al código que la llamó, esto viene dado por la palabra `void` del principio. Si en lugar de `void` hubiéramos puesto por ejemplo `int`, deberíamos devolver un número entero mediante la sentencia `return`, por ejemplo:

```
return status;
```

Bien, dentro de `muestraStatus()`, lo que hacemos es comprobar el valor de `status` (que es el mismo que el de `alarmStatus`) y dependiendo de su valor encendemos o no el led de la tarjeta; estado 0 no se enciende y estado 1 y 2 lo encendemos. Realmente podríamos haber hecho esta función sin el parámetro `status` y acceder directamente a la variable `alarmStatus`, ya que es visible dentro de la función, pero nos servirá para saber cómo pasar parámetros a las funciones.

Por último, tenemos la función `suenarAlarma()`, que lo único que hace es enviar un pulso de 200ms al altavoz para hacer la función de alarma.

Ahora, si lo ejecutamos, simplemente necesitamos poner una linterna apuntando al LDR, encender la alarma pulsando el botón y quitar la linterna o hacer sombra con la mano para que se dispare la alarma. Para apagar de nuevo la alarma solo tenemos que volver a pulsar el botón y ver que el led de la tarjeta Arduino se ha apagado indicando que la alarma está apagada.

## Piano

No vamos a pretender hacer un piano para hacer conciertos con una orquesta, pero vamos a ver como con un altavoz piezoeléctrico podemos reproducir melodías. Lo primero es introducirnos un poco en el mundo de la música a través de las notas. Las notas musicales son 7: Do, Re, Mi, Fa, Sol, La y Si, y se repiten a través de octavas (de Do a Do) para cubrir un rango de sonidos agudos hasta los graves. Una nota no deja de ser una onda y dependiendo de la frecuencia a la que se emita, corresponderá a una

nota u otra dentro de diferentes octavas. La correspondencia de las frecuencias de una octava las podemos ver en la siguiente tabla que usaremos como referencia más adelante.

<b>Nota musical</b>	<b>Frecuencia</b>	<b>Período</b>	<b>Tiempo en HIGH</b>
Do	261 Hz	3830	1915
Re	294 Hz	3400	1700
Mi	329 Hz	3038	1519
Fa	349 Hz	2864	1432
Sol	392 Hz	2550	1275
La	440 Hz	2272	1136
Si	493 Hz	2028	1014
Do	523 Hz	1912	956
Re	587 Hz	1703	851
...			

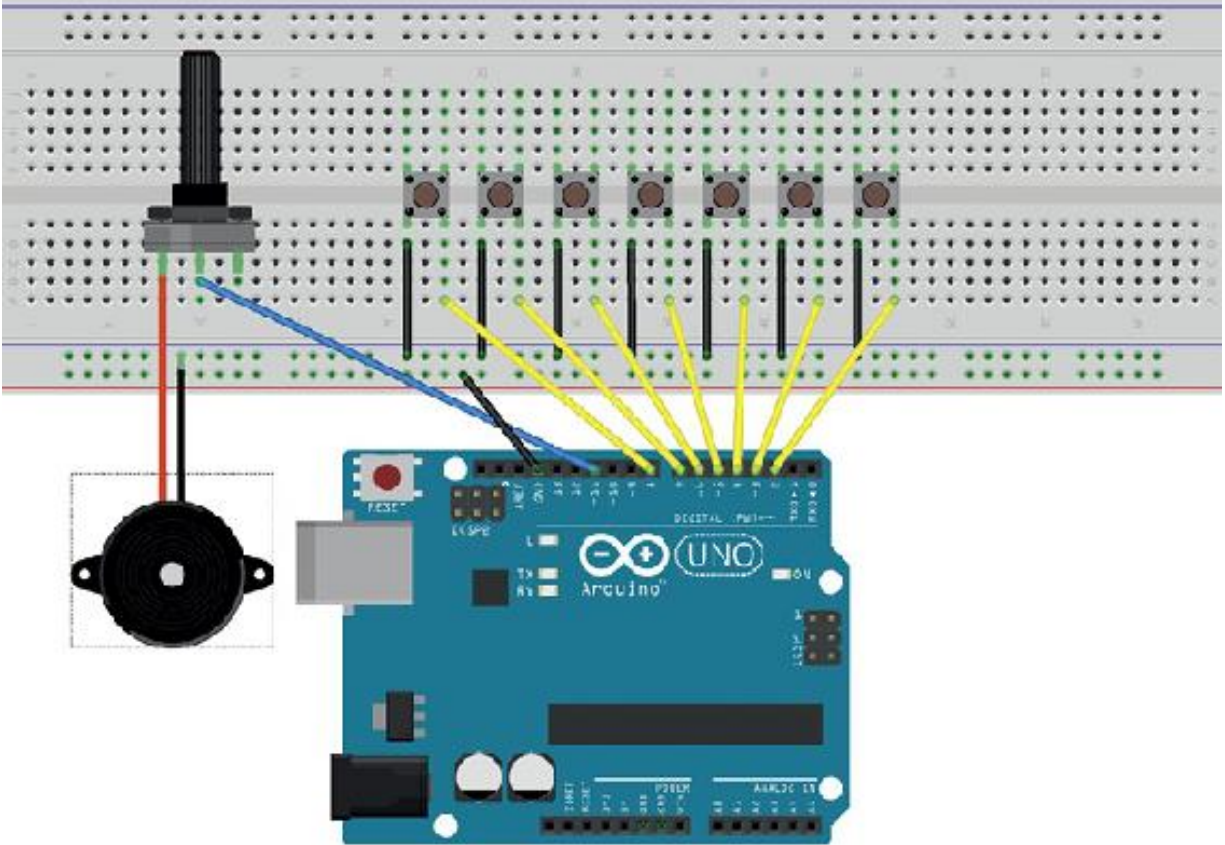
En la tabla además de las notas y la frecuencia, como datos curiosos se han añadido el período de la onda producida (que es el tiempo en milisegundos que tarda en repetirse la onda, lo que dura) y el tiempo en milisegundos que la onda se encuentra en HIGH (que es la mitad del período).

Con estos datos, si quisiéramos producir un “Re” en el altavoz, tendríamos que poner el pin al que se encontrara conectado el altavoz a HIGH durante 1700ms y otros 1700ms en LOW, y así sucesivamente el tiempo que se quiera reproducir esa nota. Está claro que podríamos hacerlo nosotros mismos utilizando la programación que ya conocemos, pero sería largo y complejo. Arduino ofrece una función llamada `tone()` que hace esto por nosotros. Esta función tiene dos o tres parámetros, el pin sobre el que enviar el pulso, la frecuencia del tono a reproducir y la duración en

milisegundos de la nota. El parámetro de duración es opcional y si no se informa se reproduce el tono constantemente hasta encontrar una instrucción `noTone()`, que hace precisamente esto, parar lo que se esté reproduciendo en ese pin. Una peculiaridad que tiene esta función es que solo es capaz de reproducir una nota a la vez, de modo que si reproducimos una nota durante 1 segundo, pero al medio segundo le decimos que cambie de nota, pasará a reproducir la segunda dejando de sonar la primera. Por ejemplo para reproducir la nota “La” durante 3 segundos sería:

```
tone(5, 440, 3000);
```

Para el circuito necesitaremos un altavoz piezoeléctrico, un potenciómetro que nos servirá de volumen y siete botones que nos servirán como teclas de piano. El uso del potenciómetro es opcional, pero dado que los sonidos que se producen con estos altavoces no agradan a todo el mundo, con el potenciómetro podremos bajar el volumen y poder trabajar sin molestar al resto de la gente de casa.



**Figura 8.4.** Circuito para piano con volumen

En el sketch tendremos que configurar cada una de las entradas usadas por los botones de modo que cuando se detecte pulsación en algún botón, se reproduzca el sonido correspondiente. Para ahorrarnos unas cuantas resistencias, los configuraremos para que trabajen en pull-up cosa que hacemos en el `setup()`.

```
const int speakerPin = 11; // altavoz
const int duration = 10; // duración
const int noteDo = 2; // DO
const int noteRe = 3; // RE
const int noteMi = 4; // MI
const int noteFa = 5; // FA
const int noteSol = 6; // SOL
const int noteLa = 7; // LA
const int noteSi = 8; // SI

void setup()
for (int i = 2; i < 9; i++) {
    pinMode(i, INPUT);
```

```

        digitalWrite(i, HIGH); // se activa la resistencia de pull-up
    }
    pinMode(speakerPin, OUTPUT);
}
void loop() {
    if (digitalRead(noteDo) == LOW) {
        tone(speakerPin, 261, duration); //reproducimos DO
    }
    if (digitalRead(noteRe) == LOW) {
        tone(speakerPin, 294 , duration); //reproducimos RE
    }
    if (digitalRead(noteMi) == LOW) {
        tone(speakerPin, 329 , duration); //reproducimos MI
    }
    if (digitalRead(noteFa) == LOW) {
        tone(speakerPin, 349 , duration); //reproducimos FA
    }
    if (digitalRead(noteSol) == LOW) {
        tone(speakerPin, 392 , duration); //reproducimos SOL
    }
    if (digitalRead(noteLa) == LOW) {
        tone(speakerPin, 440 , duration); //reproducimos LA
    }
    if (digitalRead(noteSi) == LOW) {
        tone(speakerPin, 493 , duration); //reproducimos SI
    }
}

```

En la sección `setup()` configuramos los pines para su función, el del altavoz como salida y los de los botones como entradas con resistencias pull-up. Como los botones son siete pines de entrada consecutivos, su configuración la hemos metido en un bucle en el que se incrementa el contador de 2 a 9 (funcionará hasta que el contador sea 8) y así nos es mucho más fácil en lugar de tener que hacerlos uno a uno.

La parte del `loop()` es muy sencilla, simplemente lee si está pulsado o no cada uno de los botones y en caso de estar pulsado, entonces emite su nota correspondiente. Todas las notas se reproducen de la misma forma, mediante la función `tone()` a la que le pasamos como parámetros el pin del altavoz, la frecuencia a la que debe emitir la nota (que la tomamos de la tabla de notas anterior) y la duración que tiene que tener la nota, que podemos jugar variándola para crear notas sostenidas.

Ya podemos ejecutar el `sketch` y convertirnos en el próximo Mozart.



# 9

## Actuadores



**En este capítulo aprenderá a:**

- Conocer las opciones de motores más comunes
- Usar servos eléctricos
- Hacer un paso a nivel con barrera
- Controlar el flujo de corriente mediante relés
- Encender una lámpara cuando anochezca

En los capítulos anteriores hemos sido capaces de captar situaciones del entorno de nuestra tarjeta Arduino, tales como falta de luz, que alguien pulse un botón o temperaturas. Es el momento de ver cómo actuar para modificar el entorno físico que nos rodea, por ejemplo encendiendo luces, abriendo puertas, etc.

Podemos decir que un actuador es aquel que es capaz de transformar una corriente eléctrica en una salida normalmente mecánica, por ejemplo un motor gracias a la corriente eléctrica puede girar, un pistón podría moverse, etc. Aunque son muchos los actuadores y muy distintas sus aplicaciones veremos dos de ellos, un motor y una especie de interruptor llamado relé.

## Motores

Los motores son unos actuadores capaces de transformar energía eléctrica en mecánica, haciendo que roten a partir de su eje. Existen de muchos tipos de motores algunos muy pequeños y otros muy grandes, de corriente alterna y de corriente continua, equilibrados y no equilibrados... todos tienen su aplicación y cuando tengamos que seleccionar un motor, debemos tener en cuenta para qué lo vamos a utilizar. Dos tipos de motores muy utilizados en proyectos de electrónica son los motores “paso a paso” y los servomotores:

- **Motores paso a paso:** También se les conoce como *steppers*, nos permiten controlar de manera muy precisa el giro y son bastante baratos. El giro del motor no es suave, sino que lo hacen a saltitos (de ahí su nombre). Se clasifican según el número de pasos que necesita para dar una vuelta o los grados que gira en cada paso. Por ejemplo si un motor en cada paso gira  $1.8^\circ$ , necesitaríamos  $360^\circ/1.8^\circ = 200$  pasos para completar la vuelta y si necesitaríamos que girara solo  $90^\circ$  (por ejemplo para abrir una puerta) pues tendríamos que

hacerle girar 50 pasos. La velocidad en la que se producen estos pasos la podemos controlar de modo que el giro sea más o menos rápido. Para utilizar estos motores con Arduino se necesitan unos controladores externos que nos ayuden a enviar correctamente los pulsos eléctricos al motor. El entorno Arduino contiene una librería para ayudarnos a usar estos motores, es la *Stepper* y puede accederse a ella mediante el menú Sketch>Importar librería...>Stepper o incluir su archivo de cabecera `#include <Stepper.h>`.

- **Servomotores:** También se les conoce simplemente como servos y son muy fiables a la hora de posicionarlos, fáciles de manejar y baratos. Estos motores se componen de tres partes, un motor, un potenciómetro y una caja de engranajes. Su funcionamiento explicado de manera rápida es el siguiente: cuando recibe la señal eléctrica, se mueve el brazo de control del servo a la vez que mueve el potenciómetro interno y la posición la consigue cuando se igualan las tensiones internas y las que le enviemos nosotros para posicionarlo. Vamos a ver un poco más de estos motores.

## Servomotores

Los servomotores nos van a permitir hacer girar su eje un número determinado de grados a partir de la señal que le enviemos. Si nos fijamos en la figura 9.1, podemos ver que los servomotores suelen venir acompañados de unos elementos pegados a su eje para facilitar los montajes mecánicos.

Los servomotores disponen de tres entradas, dos para alimentarlos y una tercera para recibir la señal que servirá para posicionar su eje.



**Figura 9.1.** Servomotores

El movimiento del brazo del servomotor se indica mediante pulsos eléctricos PWM (ya los hemos visto en capítulos anteriores); y cada posición del motor viene dado por un pulso diferente; por ejemplo si el pulso es de 1ms, se indica que se debe colocar a  $0^\circ$ , si el pulso es de 2ms se debe colocar a  $180^\circ$  y a partir de estos dos valores se extrapolan el resto; entonces para colocarlo a  $90^\circ$ , el pulso debería ser de 1.5ms. El ciclo completo de pulso depende de cada servo, pero normalmente son 20ms, es decir una vez acabado el pulso, tiene de tiempo hasta los 20ms para alcanzar la posición que le hayamos indicado; así, si se le ha dicho que es la posición  $0^\circ$ , se ha dado un pulso de 1ms, con lo que tendría 19ms si quisiera alcanzar la posición  $0^\circ$  y si se le dice la posición  $180^\circ$ , habríamos hecho un pulso de 2ms, con lo que nos quedarían 18ms para colocar el brazo en la posición  $180^\circ$ . Dado que tenemos el mismo tiempo para alcanzar la posición si estamos lejos del punto como si no, la velocidad de giro depende directamente tanto de la posición actual como a la que se tiene que ir. Si estoy en  $1^\circ$  y quiero ir a  $0^\circ$  tengo 19ms para cubrir  $1^\circ$ , pero si quiero ir a  $180^\circ$  tendré 18ms para cubrir  $179^\circ$  de giro. Cada 20ms podremos enviar un pulso con una nueva posición al servo, por lo que podemos cambiar de posición 50 veces por segundo.

¿Un poco lioso? Vamos a hacer un ejemplo donde utilizaremos un potenciómetro para mover el brazo del servo.

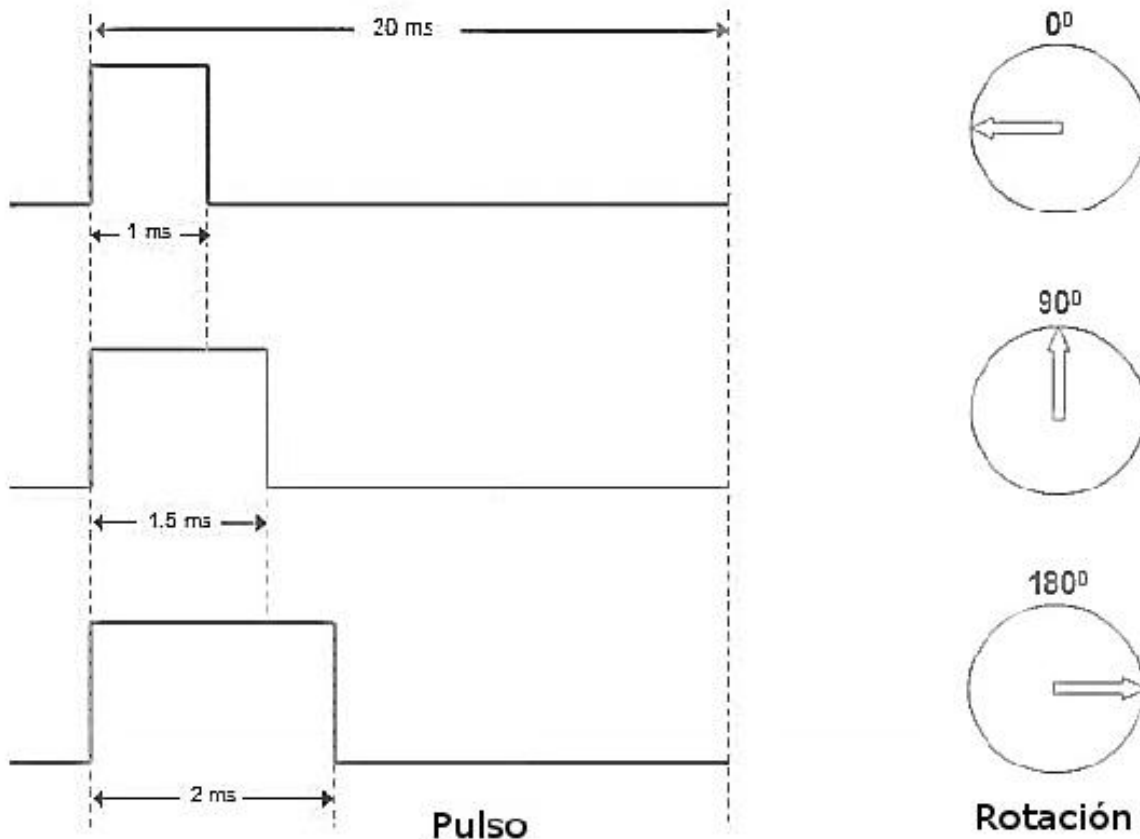


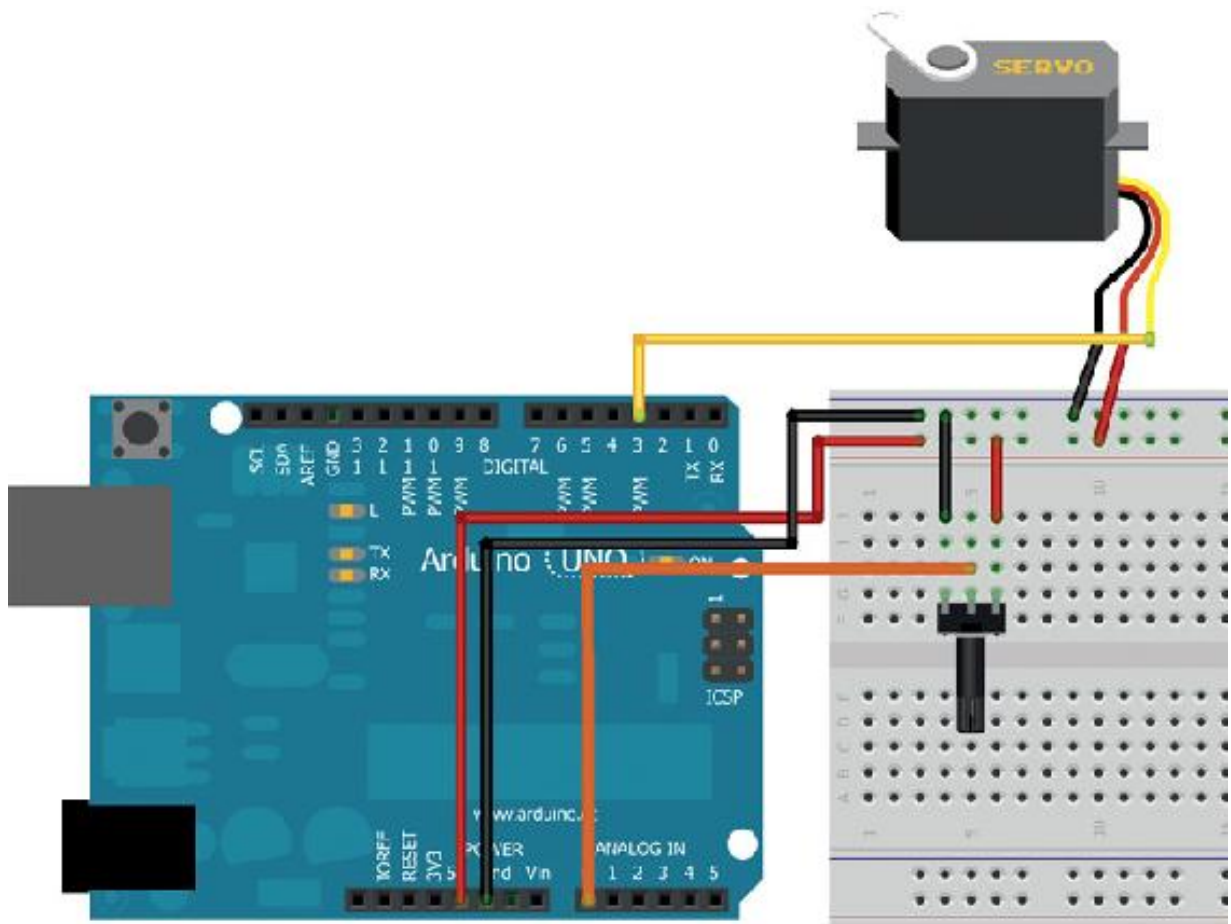
Figura 9.2. Pulsos para indicar posición

### Control del servo

En este primer ejemplo vamos a controlar un servomotor mediante un potenciómetro, de modo que podamos seleccionar la posición del servo. Como elementos que necesitaremos para realizar el circuito tenemos simplemente un servomotor y un potenciómetro. El montaje quedaría como el de la figura 9.3. Hay que tener cuidado y conectar el cable de datos a un pin de la tarjeta Arduino que tenga habilitada la salida PWM. Recordemos que estas salidas están marcadas en la propia tarjeta mediante las letras PWM o con el símbolo “~”.

Comenzamos un nuevo *sketch* y para facilitar su funcionamiento usaremos la librería lo primero que haremos es incluir la librería Servo mediante el menú Sketch>Importar librería...>Servo. Con ello tendremos unas variables de tipo *Servo* que nos facilitarán trabajar con nuestro motor. A esta variable le tendremos que indicar el pin donde se encuentra conectado

el cable de control del servo usando la llamada `attach(numeroPin)` y a partir de ese momento podremos tanto escribir la posición en la que queremos que se encuentre el servo usando la llamada a `write(grados)`, como obtener la posición en la que está en ese momento mediante la llamada a `read()`.



**Figura 9.3.** Circuito con servomotor y potenciómetro

El potenciómetro a leer lo colocaremos en la entrada analógica A0, y será quien controle la posición a la que se debe colocar el servo.

Además de leer la entrada analógica del potenciómetro, el *sketch* de este circuito va a hacer que informe en el monitor serie la posición en la que está el servo en cada momento.

Necesitaremos entonces definir el pin por el que se leerá el potenciómetro y el pin del servomotor, leer el valor del potenciómetro, mapearlo valores comprendidos entre 0° y 180° y mostrarlo en el monitor serie.

El *sketch* sería:

```
#include <Servo.h>

Servo servo; //crea la instancia de la clase

int servoPos = 0; // posición del servo
const byte servoPin = 3;
const byte potPin = A0;

void setup(){
  servo.attach(servoPin); // Se une el servo con el pin 3
  pinMode(potPin, INPUT); // el potenciómetro a A0
  Serial.begin(9600);
}

void loop(){
  int value = analogRead(potPin);
  value = map(value, 0,1023, 0,180);
  if (abs (servoPos-value) > 5){ //evitar movimientos

    Serial.print("Valor en la entrada: ");
    Serial.println(value);
    servo.write(value);
    delay(20); //tiempo de espera
    servoPos = servo.read();
    Serial.print("Estado del servo: ");
    Serial.println(servoPos);
  }
}
```

Muy importante es la entrada *delay(20)* que hay en el bucle, ya que si no estuviera, el bucle principal `loop()` se ejecutaría más de una vez en el intervalo de 20ms y recordemos que este es el tiempo que necesita el servo para acabar su ciclo; le estaríamos dando dos o más posiciones dentro del mismo ciclo y solo podría atender una.

---

**NOTA:** Un ciclo completo del servomotor se completa en 20ms.

---

La línea de código

```
if (abs (servoPos-value) > 5){
```

nos servirá para evitar que el servo se mueva continuamente. Como la posición del servo viene dada por el valor leído del potenciómetro, es posible que existan pequeñas fluctuaciones en la tensión que leemos del potenciómetro, y que sin tocarlo, en ocasiones lea por ejemplo 700 y otras 720, lo que haría que el servo se moviera todo el rato como si estuviera nervioso. Para evitarlo, usamos la línea anterior, que lo que hace es varía la posición del servo si y solo hay un cambio mayor de 5 grados en la nueva posición. Si probamos el circuito, veremos que podemos mover el brazo del servo simplemente moviendo el potenciómetro. También podemos ver su posición en el monitor serie.

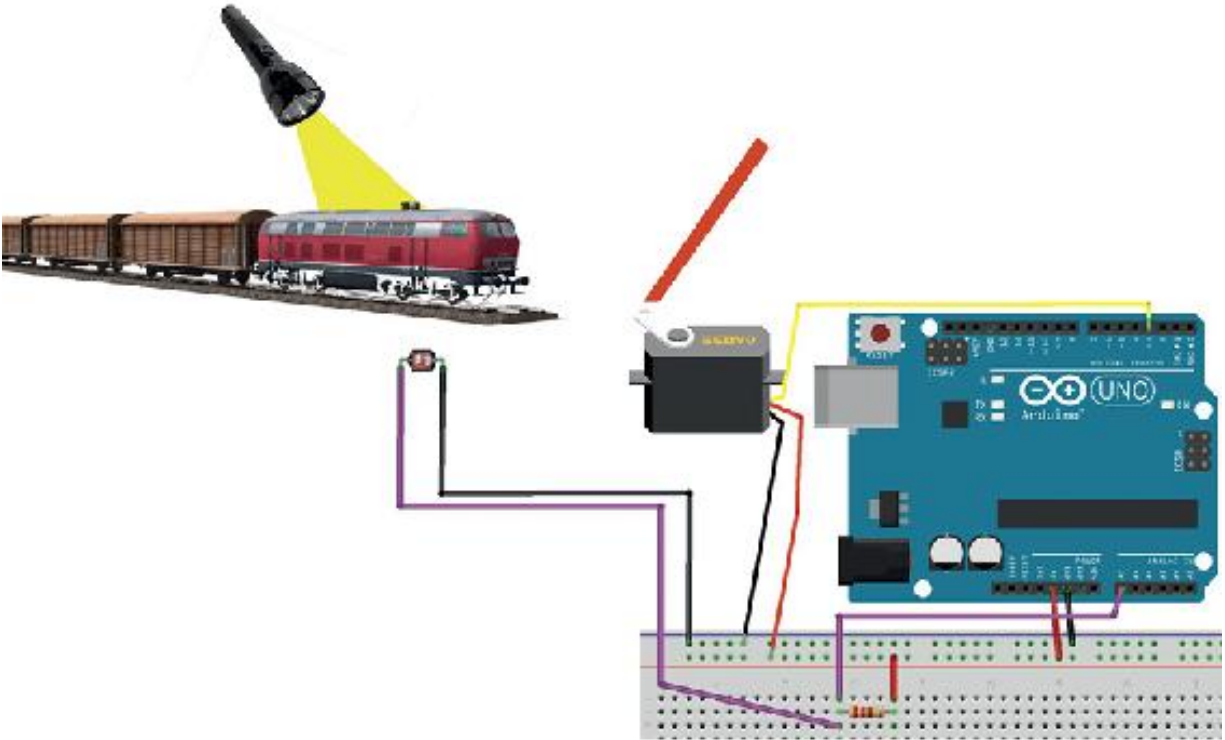
### Paso a nivel

Vamos ahora a hacer con el servo otro ejemplo que aquellos que les guste jugar con trenes podrán aprovechar... se trata de un paso a nivel con barreras. Lo que vamos a hacer un sistema que cuando detecte que se aproxima un tren, baje las barreras para que los coches no atraviesen las vías. Lo que tenemos que hacer es primero detectar que se aproxima el tren al cruce y cuando se detecte la presencia mover el servo a la posición de barrera bajada y en cuanto se aleje volver a subir las barreras.

Para la detección del tren usaremos una fotorresistencia (un LDR) que ya hemos utilizado anteriormente. Si ponemos el LDR en un lado de las vías y una luz apuntando hacia él en el otro lado de las vías, al pasar el tren producirá una sombra en el LDR que podemos utilizar para disparar el cambio de posición de las barreras. Si se quisiera, también se podría usar infrarrojos y su reacción sería más rápida, pero usaremos un LDR ya que es suficiente su tiempo de reacción y ya lo conocemos.

El circuito quedaría como el de la figura 9.4.





**Figura 9.4.** Paso a nivel

Como se puede ver, necesitaremos un LDR, una resistencia de 10K $\Omega$  para el divisor de tensión en caso de que no esté ya integrado en el LDR (recordemos que si tiene 2 patillas no tiene divisor de tensión y lo tendremos que hacer nosotros, si tiene 3 sí lo tiene), una luz (si se tiene un diodo laser, mejor, ya que se puede apuntar directamente al LDR) y un servomotor.

En este circuito, se irá leyendo en la entrada analógica los valores del divisor de tensión del LDR, y lo que tendremos que hacer es que cuando pase de un valor a otro, cambiar la posición de la barrera. Miremos cómo quedaría el *sketch*.

```
#include <Servo.h>

Servo servo; //crea la instancia de la clase

int servoPos = 0; // posición del servo
const byte servoPin = 3;
const byte ldrPin = A0;

void setup() {
```

```

servo.attach(servoPin); // Se une el servo con el pin 3
  pinMode(ldrPin, INPUT); // el LDR a A0
  Serial.begin(9600);
}

void loop() {
  int value = analogRead(ldrPin);
  Serial.print("Valor en la entrada: ");
  Serial.println(value);
  if (value > 500) { //umbral de cierre
    servo.write(90); //abrimos la barrera
  }
  else {
    servo.write(0); //cerramos la barrera
  }

  delay(20); //tiempo de espera
  servoPos = servo.read();
  Serial.print("Estado del servo: ");
  Serial.println(servoPos);
}

```

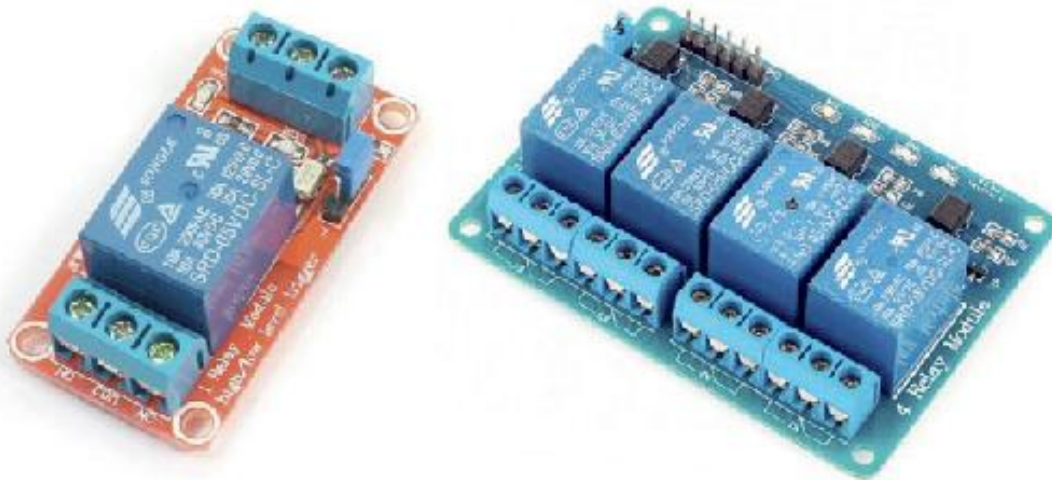
Ahora, cuando el LDR detecte una luz, el valor de entrada leído en el pin A0 será alto y cuando pase el tren, interrumpirá la luz dando sombra, momento en el cual bajará el valor obtenido en A0. Nosotros hemos puesto que para valores superiores a 500 la barrera permanezca abierta pero este valor dependerá de la luz que haya en la habitación y del tipo de luz que se esté usando en el otro lado de la vía.

Del mismo modo que hicimos con el ejemplo anterior, también vamos mostrando en el monitor serie el valor leído y la posición del servo.

## Relés

Los relés son una especie de interruptores que podemos controlar su estado mediante una señal eléctrica. Un hecho muy importante de los relés, es que el circuito de control y el circuito controlado son independientes, lo que permite que podamos tener un control con 5V en

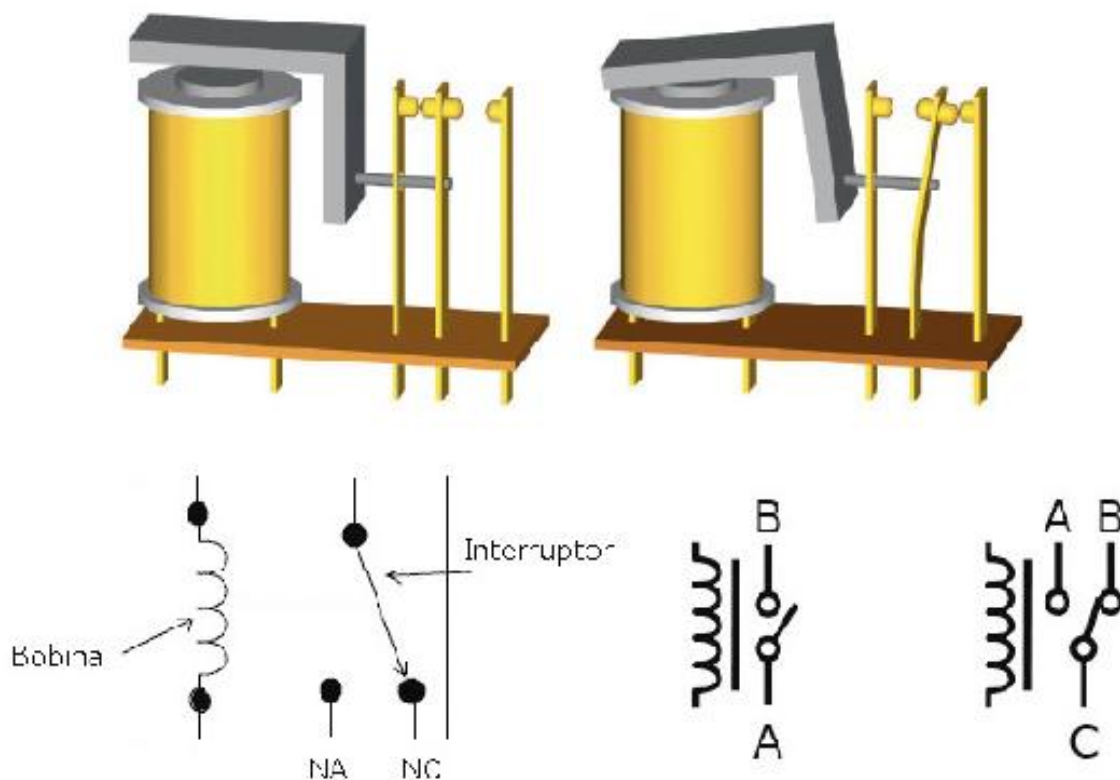
corriente continua (la de Arduino) de un circuito de 220V en corriente alterna (la de los enchufes de casa).



**Figura 9.5.** Relés

Lo usual es que el relé tenga dos o tres pines para el circuito a controlar. En caso de que tenga dos pines, puede ser que por defecto (si no hay señal en el circuito de control) esté o bien cerrado o bien abierto, y al dar la señal en el circuito de control, cambie de estado pasando de abierto a cerrado o de cerrado a abierto. Si tiene tres pines (por ejemplo A B y C), uno de ellos es común (el C) y cuando no hay señal se cierra el circuito del pin común con uno de los pines (circuito A-C) y cuando se da señal, este se abre y se cierra el circuito correspondiente al pin común y al otro pin (circuito B-C), es decir siempre hay un par de pines que nos dan circuito cerrado (encendido) y un par que dan circuito abierto (apagado). El pin que tiene el circuito cerrado en ausencia de señal de control del relé se denomina Normalmente Cerrado o NC mientras que el que está abierto en ausencia de señal se denomina Normalmente Abierto o NA (NO en inglés) y son estas marcas (NC y NO) las que suelen venir en las etiquetas de los relés; el pin que no tiene etiqueta será por descarte el común.

En el esquema de la figura 9.6 podemos verlo más claramente.



**Figura 9.6.** Esquemas de funcionamiento de un relé

Los relés los podemos encontrar en las tiendas de muchas formas y tamaños, sueltos o en grupos de varios... y también de muchas capacidades, tanto en el circuito de control como en el circuito a controlar. Por ejemplo, como trabajamos con Arduino y éste dispone de señales de control de 5V, a nosotros nos interesa seleccionar un relé que tenga un circuito de control de 5V y las características del circuito controlado dependerán de lo que queramos hacer; ahora lo veremos.

### **Control de una lámpara**

Vamos a realizar un ejemplo en el que encenderemos automáticamente una lámpara cuando el nivel de luz sea bajo. Esto nos puede servir como elemento de comodidad y no tener que levantarse a encender las luces cuando va anocheciendo o para cuando nos vamos de vacaciones, que por

la noche se encienda una luz y parezca que hay alguien en casa. Como no todos tendremos acceso a relés que soporten 220V (que es la tensión de casa) o a lo mejor nos puede dar miedo trabajar con esta cantidad de tensión, lo explicaremos también encendiendo un led.

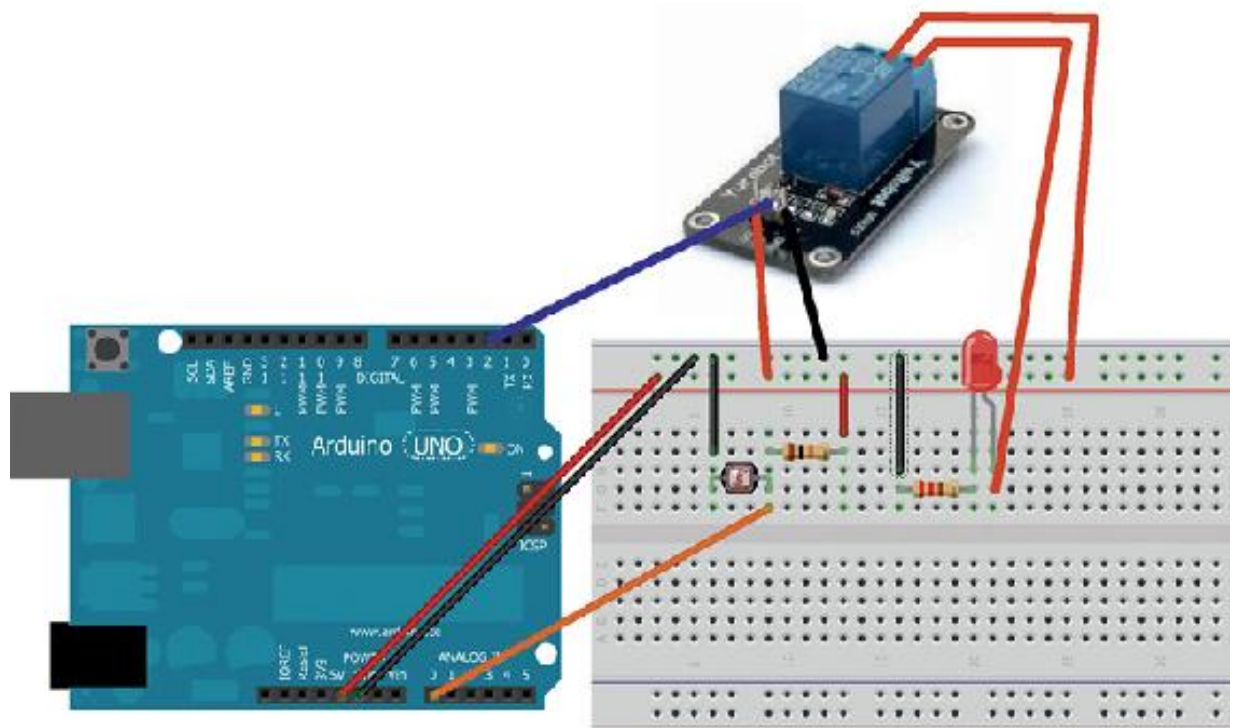
---

**IMPORTANTE:** Si se decide hacer el ejemplo con un circuito de 220V, debe hacerse acompañado de un adulto, ya que estamos trabajando con unos valores de tensión peligrosos.

---

Vamos a hacer primero el circuito con el led y luego lo transformaremos para trabajar con la lámpara. Lo primero que necesitaremos para el circuito es necesitaremos un relé, una fotorresistencia, un led, una resistencia de  $220\Omega$  para el led y otra de  $10\text{ k}\Omega$  para el divisor de tensión de la fotorresistencia. Los relés más comunes son los que disponen de tres pines o bornes en el circuito controlado, semejantes a los de la figura 9.7 así que será el que usemos.

El circuito quedaría:



**Figura 9.07.** Circuito con relé

El control del apagado y encendido de la bombilla lo tendrá la tensión que leamos en la entrada de la fotorresistencia, es decir, llegado un nivel de tensión en esa entrada, tendremos que cambiar el estado del relé para encender o apagar la luz. En el circuito controlado del relé, en este ejemplo hemos puesto un led de modo que una de sus patillas está conectada a la entrada NA (normalmente abierto) en lugar de estar directamente conectado a una entrada de la tarjeta. Esto lo que hará es que cuando desde la tarjeta se envíe la señal al relé, se cerrarán las entradas NA y común del relé y se encenderá el led.

El *sketch* sería:

```
const byte ldrPin = A0;
const byte relayPin = 3;

void setup() {
  Serial.begin(9600);
  pinMode(ldrPin, INPUT);
  pinMode(relayPin, OUTPUT);
}

void loop() {
  int value = analogRead(ldrPin);
  Serial.print("Lectura : ");
  Serial.println(value);
  if (value > 800){
    digitalWrite(relayPin, HIGH);
  }
  else{
    digitalWrite(relayPin, LOW);
  }
  delay(500);
}
```

Se puede ver que el *sketch* es muy sencillo y que lo único que tenemos que hacer en el bucle principal, es ir leyendo la entrada correspondiente a la fotorresistencia y cuando supera un valor (el que nosotros consideremos), envía una señal *HIGH* para que el relé se cierre y luzca el led y cuando el valor sea menor, entonces enviar al relé un *LOW* para abrir de nuevo el circuito y que se apague el led.

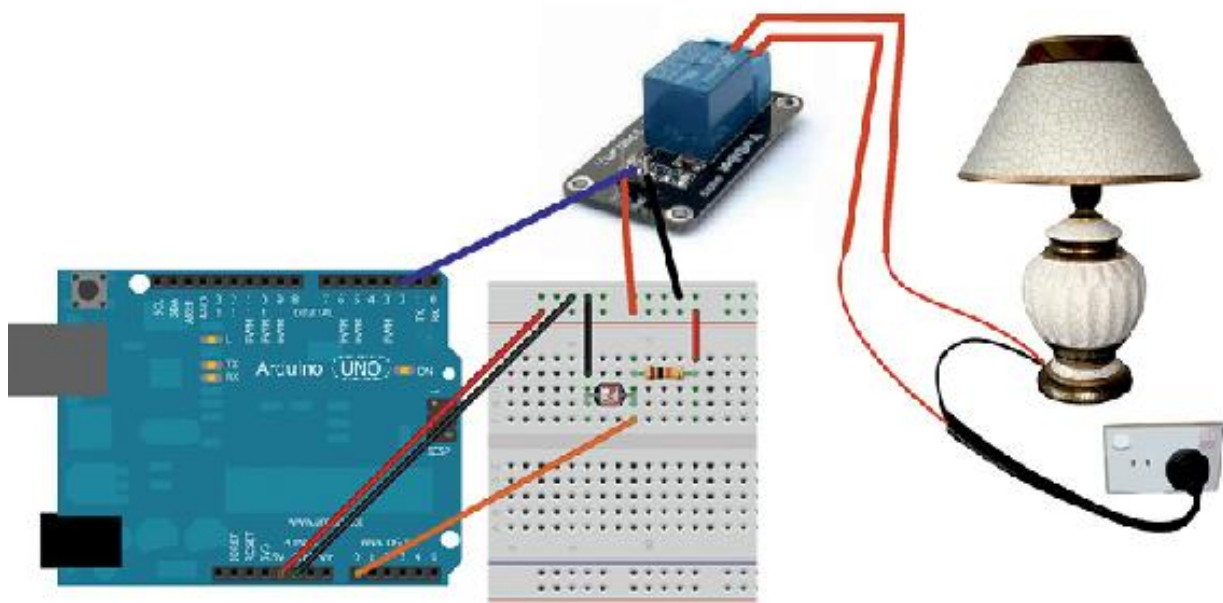
Variar este montaje para trabajar con una lámpara real es muy sencillo. Lo primero que tenemos que hacer es ver que nuestro relé soporta la tensión y la corriente que le vamos a aplicar. Cualquiera que soporte los 220V nos servirá en este caso.

---

**ATENCIÓN:** Hay que mirar el tipo de relé que tenemos y qué tensión y amperios soporta. Si usamos uno de 12V con 220V lo estropearemos y podemos hacernos daño.

---

Para incorporar la lámpara al circuito, tenemos que hacer como con el led, interrumpir uno de sus cables conectando un extremo al borne NA y otro al común, de modo que cuando se dé la señal al relé, se cerrará el circuito y la lámpara funcionará.



**Figura 9.8.** Circuito con lámpara

# 10

## Internet



### **En este capítulo aprenderá a:**

- Conocer los tipos de conexiones disponibles
- Obtener datos del tiempo de una ciudad
- Consumir servicios de Internet en general
- Enviar mensajes a Twitter
- Hacer disponibles los datos de Arduino a través de Internet



En ocasiones es necesario poder acceder a nuestra tarjeta Arduino desde lugares remotos, donde la comunicación serie o Bluetooth por ejemplo no son válidas; para ello, Arduino dispone de múltiples *shields* para ofrecernos distintas posibilidades de conexión a Internet.

De entre todas las posibles, quizá la más sencilla sea mediante cable RJ45 (el cable de red normal), aunque si necesitamos más movilidad y no estar atados a un cable, podemos utilizar conexiones WiFi o conexiones telefónicas (GSM, 3G, 4G<sup>lte</sup>). Para utilizar estas conexiones debemos utilizar la *shield* que más nos convenga, ya que en el mercado hay muchas disponibles que ofrecen uno o varios métodos de conexión, por ejemplo cable y WiFi juntos.

Sea cual sea el método utilizado para la conexión, deberemos utilizar las librerías correspondientes ya que la gestión de las conexiones es muy complicada.

Cuando se utiliza un servicio de Internet, existe una relación entre los dos actores que intervienen en la conexión; esta relación siempre es una relación cliente-servidor, es decir uno de los dos agentes ofrece un servicio (servidor) y el otro se encarga de usarlo (cliente). Durante las conexiones, Arduino puede trabajar como cliente y como servidor, es decir podemos acceder a datos que estén en internet (por ejemplo una web con información del tiempo) u ofrecer nosotros los datos (por ejemplo dejando que se conecten a nuestra tarjeta utilizando un navegador web).

## **Acceso a webs**

Cuando vamos a acceder a datos que se encuentran en una web, estamos actuando como clientes y lo que haremos es consumir servicios de ese servidor. Para facilitarnos estas tareas comunes, Arduino dispone de la librería Ethernet que nos ayuda tanto si actuamos de cliente como si actuamos de servidor. Además de importar la librería Ethernet en nuestros

*sketchs*, debemos incluir también la librería SPI (*Serial Peripheral Interface*, Interfaz de Periféricos Serie), ya que se usa internamente.

Para comprender mejor su uso vamos a crear un programa que se conecte a una web donde podemos consultar el tiempo que hará en un determinado lugar.

Lo primero que necesitaremos es una *shield* Ethernet para nuestra tarjeta Arduino. Hemos elegido la modalidad Ethernet de cable por ser la más sencilla, pero si se dispone de WiFi, el programa será muy semejante salvo que hay que introducir los parámetros de conexión como el nombre de la red y clave; podemos ver un ejemplo más adelante en el capítulo.



**Figura 10.1.** *Shield* Ethernet

Comenzaremos con un nuevo proyecto y tal y como habíamos avanzado, necesitaremos las librerías SPI y Ethernet, que pueden ser incluidas mediante el menú Sketch>Importar librería...>Ethernet y luego Sketch>Importar librería...>SPI o manualmente escribiendo las líneas:

```
#include <SPI.h>
#include <Ethernet.h>
```

En la función *setup()* tenemos que configurar la conexión de nuestra tarjeta de red; esto se hará con la llamada:

```
Ethernet.begin(mac, ip, dns, gateway, subred);
```

No vamos a entrar a ver qué es cada parámetro, ya que se necesitan unos conocimientos de red y en la actualidad, la mayoría de las instalaciones de red que tenemos en las casas no hacen necesario informarlos. El que si necesitamos informar obligatoriamente es el primer parámetro; este parámetro es la identificación única de la tarjeta de red y se denomina MAC (*Media Access Control*); muchas veces viene una pegatina en el dispositivo con este número, pero si no, podemos inventarnos uno siempre que no coincida con el número MAC de algún dispositivo que ya tenemos conectado (si nos inventamos uno, es casi imposible que coincida con otro). El número MAC se da mediante un *array* de 6 elementos en hexadecimal.

Los dispositivos conectados a Internet, necesitan un número de identificación que se utiliza para saber quién se está comunicando con quién (a modo de número de teléfono), si no indicamos ninguno en el parámetro *ip*, nuestra red nos dará uno automáticamente usando un protocolo llamado DHCP. Del mismo modo, este protocolo nos informara automáticamente el resto de parámetros necesarios para la conexión.

El esqueleto del *sketch* para trabajar con la tarjeta de red sería:

```
#include <SPI.h>
#include <Ethernet.h> //número mac inventado byte mac[] = { 0xAA, 0xAB, 0xAC, 0xAD,
0xAE, 0xAF }; void setup(){ Ethernet.begin(mac); } void loop () {}
```

Este *sketch* no es muy funcional, pero podemos ya obtener por ejemplo la dirección *ip* que nos haya dado nuestra red mediante `Serial.println(Ethernet.localIP())` (tenemos que configurar primero el monitor serie para que funcione ese comando).

Vamos a avanzar un poco más y traer datos de una página web.

Para trabajar como cliente, necesitamos una variable de la clase *EthernetClient*, que nos ayudará a crear las conexiones de manera sencilla, por ejemplo:

```
EthernetClient client;  
client.connect(servidor,puerto);
```

Donde necesitamos informar dos parámetros: `servidor` que es el nombre o la dirección IP del servidor que nos queremos conectar y el `puerto` que es el puerto donde está el servicio al que nos queremos conectar. Aunque suene un poco raro esto de direcciones IP, puertos y demás, no tenemos que preocuparnos si no lo entendemos, ya que son cosas de conexiones de Internet que quedan totalmente fuera del temario del libro, pero que al menos nos suene.

Una vez el cliente esté conectado, podemos usarlo para realizar tanto lecturas como escrituras sobre él.

Para ver mejor el funcionamiento, vamos a utilizar el siguiente *sketch* para obtener información meteorológica de una página web (concretamente de la ciudad de Santander en la web [api.openweathermap.org](http://api.openweathermap.org)):

```
#include <SPI.h>  
#include <Ethernet.h>  
  
byte mac[] = { 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF };  
IPAddress server(188,226,175,223); //api.openweathermap.org  
  
EthernetClient client;  
boolean ethernetEnabled = false;  
  
void setup() {  
  Serial.begin(9600);  
  
  // Iniciamos la Ethernet  
  if (Ethernet.begin(mac) == 0) {  
    Serial.println("No ha funcionado el DHCP");  
    return ;  
  }  
  // Tiempo de seguridad para terminar la inicialización de la tarjeta  
  delay(1500);  
  Serial.println("Se comienza la conexión...");  
  if (client.connect(server, 80)) {  
    Serial.println("Conectados!");  
  }  
}
```

```

    // Petición HTTP
    client.println("GET /data/2.5/weather?id=3109718"); //datos de Santander
client.println();
    //cambiamos el valor de la variable para saber que está activa la Ethernet
    ethernetEnabled = true;
}
else {
    // fallo en la conexión
    Serial.println("Ha fallado la conexión");
    return;
}
}

void loop(){
    //si hay caracteres a leer en la conexión, los volcamos al monitor
    if (client.available()) {
        char c = client.read();
        Serial.print(c);
    }

    // Si se cierra la conexión, acabamos de desconectarnos del servidor
    if (!client.connected()) {
        Serial.println();
        Serial.println("Desconectados");
        client.stop();
        for(;;){}
    }
}
}

```

Si conectamos una tarjeta *shield* Ethernet a nuestra Arduino y tras conectar el cable de red a la tarjeta ejecutamos el programa, veremos en el monitor serie la información meteorológica de Santander y es una salida en un formato llamado JSON que será semejante a:

Se comienza la conexión...

Conectados!

```

{"coord":{"lon":-3.8,"lat":43.46},"sys":
{"message":0.0111,"country":"ES","sunrise":1435120515,"sunset":1435176009},"weather":
[{"id":800,"main":"Clear","description":"Sky is Clear","icon":"01d"}],"base":"cmc
stations","main":
{"temp":282.886,"temp_min":282.886,"temp_max":282.886,"pressure":973.48,"sea_level":103
2.78,"grnd_level":973.48,"humidity":91},"wind":{"speed":1.01,"deg":167.004},"clouds":
{"all":0},"dt":1435124696,"id":3109718,"name":"Santander","cod":200}

```

Desconectados

En el *sketch* lo primero que hacemos es iniciar la Ethernet con `if (Ethernet.begin(mac) == 0)` comprobando que se inicialice bien, luego nos conectamos al servidor web `if (client.connect(server, 80))` y si todo ha ido correcto enviamos una serie de parámetros que harán que se descargue el tiempo en Santander. La sección más importante es

```
client.println("GET /data/2.5/weather?id=3109718"); //datos
de Santander
```

ya que se le indica la dirección dentro del servidor a la que se quiere conectar. Dentro de `loop()` lo que hacemos es leer y mostrar en el monitor serie cara carácter que se nos envía desde el servidor. Una vez que el servidor termina de enviarnos información, podemos acabar la comunicación y desconectar el cliente mediante `client.stop()`. Un elemento extraño en el sketch es la sentencia:

```
for(;;){}
```

ésta es un bucle infinito, es decir se repetirá siempre esa instrucción. Esto nos sirve para hacer que el programa solo se ejecute una vez, puesto que cuando llegue a esta instrucción, no ejecutará ninguna más. Para conseguir parar el programa, se puede usar esta sentencia u otras como:

```
do {} while (1);
```

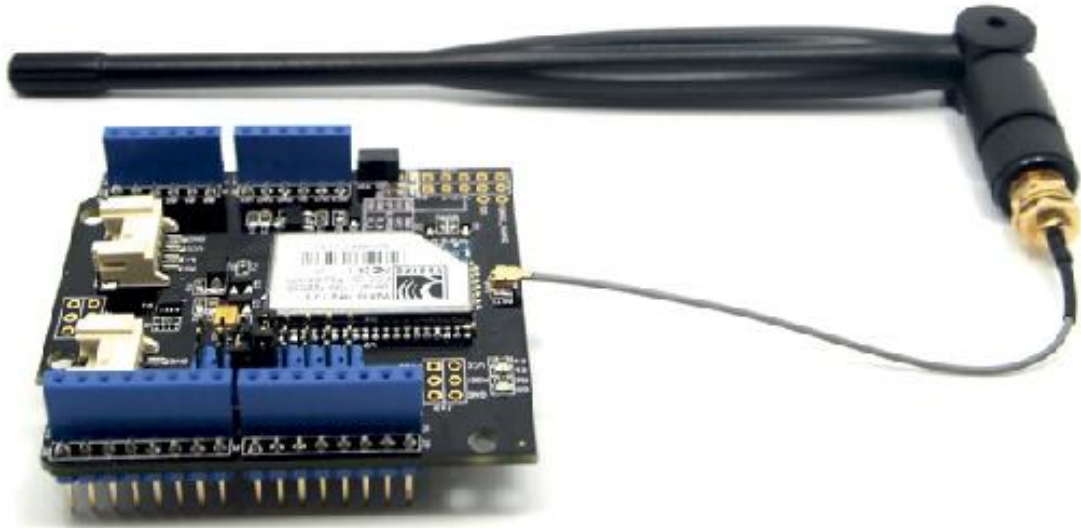
## Otras conexiones

Como hemos dicho, además de con la tarjeta Ethernet, podemos conectarnos a internet mediante WiFi y acceso telefónico. Estas conexiones funcionan de manera similar, simplemente cambiando la forma en la que se configura la conexión. Si se utilizan librerías externas, hay que leer la información que viene acompañada con ellas. Veamos rápidamente como deberíamos configurar la conexión en alguno de los casos.

## WiFi

Para poder trabajar con conexiones WiFi, lo primero que debemos hacer es importar la propia librería WiFi en el *sketch* mediante el menú Sketch>Importar librería...>WiFi o bien añadiendo la línea de código:

```
#include <WiFi.h>
```



**Figura 10.2.** *Shield WiFi*

Lo que tiene de particular una conexión WiFi, es que necesitamos el nombre de la red a la que conectarnos y una clave válida (ya que normalmente estas redes están protegidas). El identificador de la red también se conoce como SSID (*Service Set Identifier*, Identificador de Conjunto de Servicios). La conexión la haremos mediante la línea:

```
WiFi.begin(ssid, clave);
```

Indicando el identificador de red y la clave. Así, el esqueleto del *sketch* quedaría:

```
#include <WiFi.h>
char ssid[] = "SSIDDeMiRed"; //SSID
char pass[] = "miClaveSecreta"; // clave WPA
void setup(){
  Serial.begin(9600);  if (WiFi.begin(ssid, pass)== WL_CONNECTED){
    Serial.print("Conectado!");
  }
}
```

```
void loop {  
  //código que trabaja con WiFi  
}
```

## GSM

Si utilizamos conexiones GSM, en lugar de una clave, necesitaremos desbloquear la tarjeta con el número pin, para que se pueda conectar, tal y como hacemos en el teléfono móvil. También utilizaremos una librería, la GSM, que podemos introducir la línea de código:

```
#include <GSM.h>  
o hacerla accesible mediante el menú Sketch>Importar librería...>GSM.
```

Para desbloquear la tarjeta mediante pin, utilizaríamos la llamada a la función:

```
gsm.begin(numpin)
```

donde `numpin` en este caso es el número pin secreto de la tarjeta de teléfono. El esqueleto del *sketch* quedaría:

```
#include <GSM.h>  
define PIN_NUMBER "1234" GSM gsm;  
void setup(){  Serial.begin(9600);  
  boolean notConnected = true; //estado de la conexión  
  //esperar hasta conectarse  while(notConnected) {  
    if(gsm.begin(PIN_NUMBER)==GSM_READY){  notConnected = false;  
    }  
    else {  Serial.println("No se ha podido conectar");  delay(1000);  }  }  
  //Se ha podido conectar  Serial.println("Conectado!"); }
```





Figura 10.3. *Shield GSM*

Si optamos por este tipo de comunicación, podemos además acceder a las funciones típicas de un teléfono, tales como realizar llamadas o enviar SMS. ¿No es interesante poder hacer una alarma que nos envíe un SMS si detecta algo?

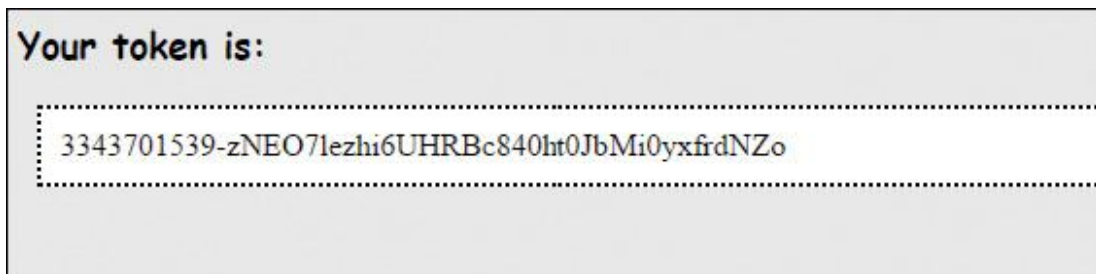
## Publicación Twitter

Una de las redes sociales más famosas del momento es Twitter. Vamos a ver cómo podríamos publicar mensajes desde Arduino, así, podríamos tener la estación meteorológica que hicimos anteriormente, que en lugar de mostrar la información en el monitor serie, la publicara en Twitter.

Lo primero que haremos es crearnos una cuenta en Twitter si no la tenemos, para ello nos vamos a <http://twitter.com>, nos registramos y validamos el correo electrónico que nos envían.

Para facilitarnos la publicación, vamos a valernos de una página web, que nos creará una cadena de texto (llamada *token*) que hace que esa web tenga acceso la cuenta creada; es lo que se denomina autenticación OAuth. En la

web <http://arduino-tweet.appspot.com> pulsamos sobre el enlace **Get a token to post a message using OAuth**.(Obtener un *token* para enviar mensajes usando OAuth) e introducimos el usuario y clave de nuestra cuenta para dar permiso a esta web a publicar siempre que usemos el *token*.



**Figura 10.4.** *Token* proporcionado

Disponemos también de unas librerías que facilitarán la tarea engorrosa de manejar la autorización con el *token* y la publicación en si dentro de Twitter. Estas librerías las podemos descargar desde <http://arduino-tweet.appspot.com/Library-Twitter-1.3.zip> o desde <http://playground.arduino.cc/Code/TwitterLibrary>; y una vez descargadas las debemos descomprimir y colocar dentro del directorio `Mis Documentos\Arduino\libraries\` en Windows. Todo esto se debe hacer con el entorno Arduino sin estar abierto o bien reiniciarlo tras colocar las librerías.

El *sketch* sería el siguiente:

```
#include <SPI.h>
#include <Ethernet.h>
#include <Twitter.h>

Twitter twitter("3343701539-zNEO7lezhi6UHRBc840ht0JbMi0yxfrdNZo"); // cambiar el token
byte mac[] = { 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF };
char msg[] = "Hola desde Arduino"; // el mensaje a enviar
void setup() {
  delay(30000);
  Ethernet.begin(mac);
  Serial.begin(9600);
}

void loop() {
```

```

Serial.println("Conectando ...");
if (twitter.post(msg)) {
int status = twitter.wait();
if (status == 200) {
Serial.println("Conectado.");
} else {
Serial.print("Error : codigo ");
Serial.println(status);
}
} else {
Serial.println("Error en la conexión.");
}
do {} while (1); //bucle infinito
}

```

Lo primero que hacemos es crear una variable de tipo `Twitter` dándole como parámetro el *token* que hemos recibido al registrarnos en la web, el que aparece en la figura 10.4 (no hemos de escribir el que aparece en esa figura, sino que cada persona que se registra recibe uno diferente). La configuración de la conexión Ethernet es igual que la realizada anteriormente. En el bucle principal, realizaremos el envío de los datos mediante `twitter.post(msg)` y esperaremos el estado de la conexión, que recuperamos mediante `int status = twitter.wait()`. En caso de recibir un estatus distinto a 200 (que significa que todo ha ido bien), lo mostramos en el monitor serie para saber qué está pasando y poder comparar el error con la lista de errores posibles que está disponible en <http://dev.twitter.com/docs/error-codes-responses>.

Un error muy común es el 403, que se suele dar por no haber escrito bien el *token* o por no cumplir las reglas de Twitter, que son:

- No más de un mensaje por minuto
- No repetir mensaje.

Para evitar que el bucle principal se repita enviando muchos mensajes, lo que hacemos es crear un bucle infinito para que se detenga ahí el programa; esto se consigue mediante:

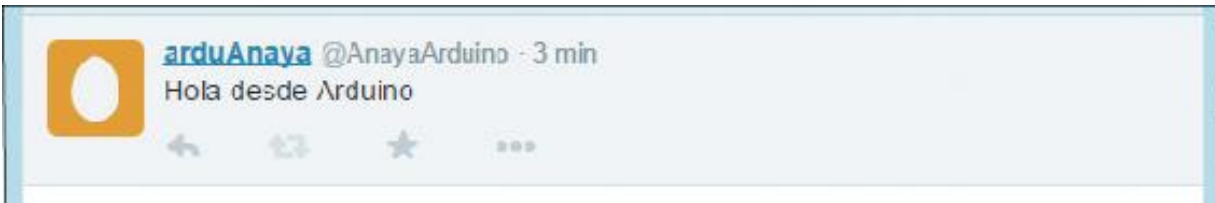
```
do {} while (1); //bucle infinito
```

---

**ATENCIÓN:** Por norma de Twitter, podemos enviar un máximo de un mensaje por minuto y cada mensaje debe ser único; de lo contrario nos dará un error 403.

---

Si ejecutamos el *sketch*, se realizará una publicación que podemos ir a consultar a la misma página de Twitter, como muestra la figura 10.5.



**Figura 10.5.** Publicación en Twitter

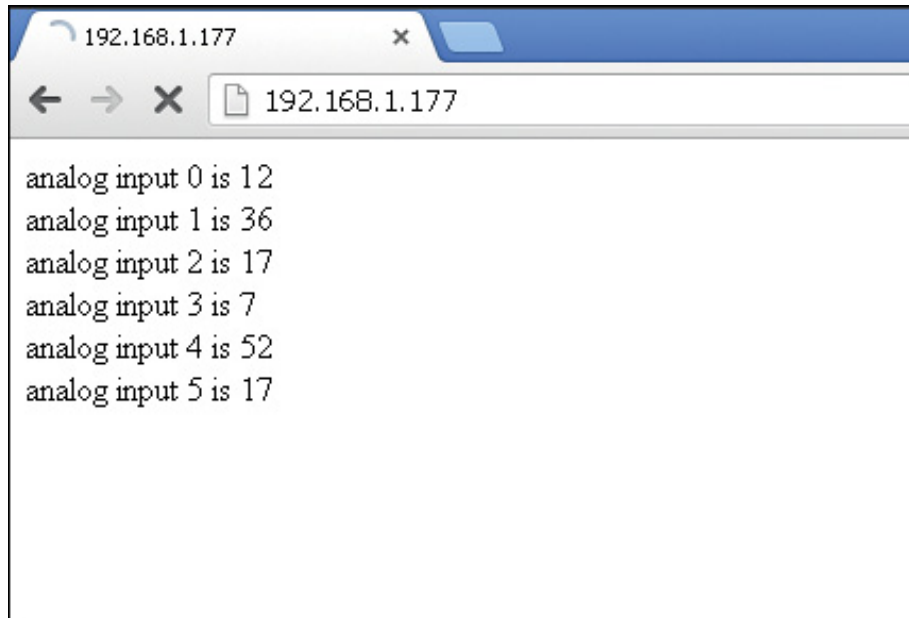
## Arduino como servidor

Además de como cliente, Arduino permite actuar como servidor en las conexiones. Actuar como servidor significa que tenemos que ofrecer un servicio en un puerto y estar escuchando a la espera de que un cliente se conecte.

No vamos a ver un ejemplo desde cero, para este caso vamos a cargar un ejemplo disponible ya en el entorno Arduino, para ello pulsamos el menú Archivo>Ejemplos>Ethernet>WebServer (no incluiremos el código en el libro ya que lo podemos ver fácilmente en pantalla). Este ejemplo se trata de un servidor web que dará las lecturas de todas las entradas analógicas. Si lo ejecutamos y nos conectamos desde el navegador web de un ordenador que esté en la misma red que la tarjeta Arduino a la dirección 192.168.1.177 veremos una salida semejante a la de la figura 10.6.

Vamos a ver un poco el código del *sketch*. Lo primero a tener en cuenta es que vamos a darle un número de IP fijo para podernos conectar al servidor. Esto se hace en la línea:

```
IPAddress ip(192, 168, 1, 177);
```



**Figura 10.6.** Resultado del ejemplo mostrado en el navegador

Si queremos usar otra dirección no tenemos más que cambiarla y poner el número deseado. Se configura el servidor para que trabaje en el puerto 80, que es el puerto habitual del protocolo HTTP utilizado por los navegadores web.

```
EthernetServer server(80);
```

En cuanto a la función `setup()` nada nuevo, pero sí en la `loop()` donde tenemos que ir mirando si hay algún cliente que quiera utilizar nuestro servicio. Para ver si hay algún cliente esperando usamos:

```
EthernetClient client = server.available();
```

Si existe, hay que mirar si tiene algún dato para ser leído de su petición.

```
if (client) {  
  while (client.connected()) {  
    if (client.available()) {  
      char c = client.read();  
      ...  
    }  
  }  
}
```

Además tenemos que saber cuándo se produce el final de la petición para comenzar a mostrar los datos. El funcionamiento es el siguiente: el

servidor irá leyendo en los datos enviados por el cliente, buscando una línea sin contenido que indique el final de la petición (parte del estándar HTTP usado por los navegadores web), o dicho de otra forma, cada vez que detecta un “\n” debe inicializar la variable *currentLineIsBlank* indicando que es nueva línea, si además de ser *currentLineIsBlank* verdadero, recibe otro “\n” quiere decir que se ha acabado la petición y podemos comenzar con la respuesta.

```
if (c == '\n' && currentLineIsBlank) {
```

La respuesta además de los datos obtenidos en la tarjeta se deben devolver en un formato llamado HTML que es lo que entienden los navegadores y además de los datos de la tarjeta se deben enviar ciertas cadenas de texto para que el navegador sepa que le enviamos datos a mostrar. La parte donde realmente se leen las entradas analógicas y se añaden a la salida a mostrar en el navegador es:

```
client.println("<html>");
client.println("<meta http-equiv=\"refresh\" content=\"5\">");
for (int analogChannel = 0; analogChannel < 6; analogChannel++) {
  int sensorReading = analogRead(analogChannel);
  client.print("analog input ");
  client.print(analogChannel);
  client.print(" is ");
  client.print(sensorReading);
  client.println("<br />");
}
client.println("</html>");
break;
```

Donde mediante un bucle los recorreremos, obtenemos su valor y lo añadimos a la salida. Una vez enviada la página web al navegador, se cierra la conexión con él.

```
client.stop();
```

La información del navegador se irá refrescando automáticamente cada 5 segundos, esto lo hemos conseguido con el comando `meta` incluido en la salida generada por Arduino.

Ahora con lo que ya sabemos, podríamos modificar este *sketch* y hacer que en lugar de dar simplemente las lecturas de las entradas analógicas, pudiera dar la temperatura medida por un termistor conectado a una de estas entradas. Sencillo ¿no?

Edición en formato digital: 2016

© Ediciones Anaya Multimedia (Grupo Anaya, S. A.), 2016  
Calle Juan Ignacio Luca de Tena, 15  
28027 Madrid

ISBN ebook: 978-84-415-3778-1

Todos los nombres propios de programas, sistemas operativos, equipos hardware, etc. que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Está prohibida la reproducción total o parcial de este libro electrónico, su transmisión, su descarga, su descompilación, su tratamiento informático, su almacenamiento o introducción en cualquier sistema de repositorio y recuperación, en cualquier forma o por cualquier medio, ya sea electrónico, mecánico, conocido o por inventar, sin el permiso expreso escrito de los titulares del Copyright.

Conversión a formato digital: calmagráfica

[www.anayamultimedia.es](http://www.anayamultimedia.es)