Desarrollo práctico con Arduino

Joan Ribas Lequerica

A mi hija Mar, gracias por tus sonrisas contagiosas
y por recordarme cada día lo increíble que es investigar y descubrir cosas nuevas;
espero que nunca pierdas ni la ilusión por aprender, ni la sonrisa.

Agradecimientos

A mi mujer Laia por ser como es.
A mi sobrino Pablo, espero estar a la altura ejerciendo de padrino.
A mi familia natural y política por... todo; creo que soy afortunado.
A todos aquellos que comparten su conocimiento y hacen que los demás podamos aprender.
A los que hoy hayan ayudado a alguien.
A ti, si aguantas hasta el final del libro.

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. Es autor de numerosas publicaciones en revistas tecnológicas y libros, entre los que se encuentran "Programación en Delphi", "Guía práctica Web Services", "Manual imprescindible de Desarrollo de aplicaciones para Android", "Manual imprescindible de Desarrollo de aplicaciones para Android. Edición 2013" y antiguo colaborador de proyectos GNU como Gentoo o Enlightenment.

Índice de contenidos

Introducción

Durante el proceso de producción de artículos electrónicos, existen varias fases que hacen posible que la idea conceptual en la cabeza de algún ingeniero pueda convertirse en el producto final, producto que luego encontraremos en las tiendas. Aunque obviamente el número de fases depende de la complejidad del proyecto que se quiera llevar a cabo, a groso modo podríamos nombrar las más importantes, valiéndonos para ello de la idea del semáforo:

- Concepción de 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 que 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 extraer 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 carriles, simplemente esto, ya se verá qué tipo de señal debe mostrar o cuando, pero lo que se quiere es algo que muestre si se puede pasar o no.
- Modelado de la abstracción: Consiste en dar una solución que satisfaga la abstracción realizada sobre la idea. Puede que diversos modelados den esta solución y que haya que seleccionar el más óptimo, 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... suponga que decide modelarse mediante 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. También dependiendo de las situaciones se deberá optar por unas opciones u otras. Si se miran los pasos a nivel del tren, son simplemente unos semáforos, pero se les ha añadido unas barreras mecánicas (en algunos casos aún se pueden ver pasos a nivel sin barrera) para dificultar el paso cuando está en rojo, ya que el coste de pasar el semáforo en rojo es muy alto.
- Diseño del prototipo: Se realiza un diseño del prototipo para implementar el modelo que se ha elegido para dar solución al problema. En el caso que se expone, previos cálculos de corrientes y tiempos, se debe crear una caja con luces, un controlador que sepa qué luces se deben encender en cada momento y orquestarlas de modo que en el cruce no se puedan encontrar todos los semáforos en verde a la vez. El prototipo servirá para comprobar que durante los pasos de abstracción de la idea y modelado

no se han olvidado premisas o casos por satisfacer, por ejemplo si se ha pensado el semáforo sólo 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 muchos costes en el producto final... Esta fase es de vital importancia 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 software que sea capaz de colocar los elementos necesarios del circuito (resistencias, condensadores, pulsadores...) en una placa y crear las rutas de las pistas de unión entre ellos (los cables de nuestro prototipo) de modo que no se crucen. Dependiendo de la complejidad del proyecto, los cálculos a realizar para que no se crucen las pistas son muy complejos e inviables de realizar manualmente, en ocasiones, la única manera de realizar las conexiones sin que las pistas se crucen es creando placas multicapa, donde en cada capa se dibujan unas pistas de unión entre componentes. 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.

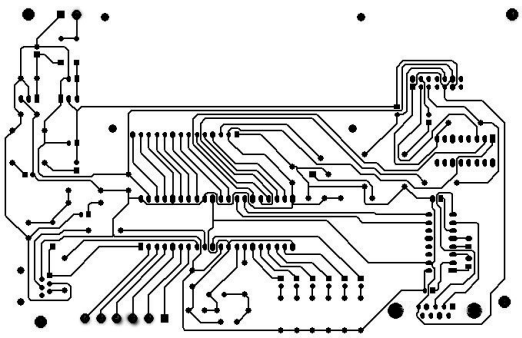


Figura 1. Fotolito de un circuito.

- Unión de componentes: Cuando ya se tiene la placa creada a partir de los fotolitos (denominada también PCB del inglés *printed circuit board* o tarjeta de circuito impreso), lo que realmente se tiene es una superficie no conductora (realizada en materiales como baquelita, resinas de fibra de vidrio reforzada, teflón, plástico...) sobre la que se han "dibujado" las pistas conductoras habitualmente realizadas en cobre. Sobre estas pistas será donde se fijen los componentes necesarios que conformen el circuito final ya montado. Sobre estas pistas pueden existir unos agujeros; esto es debido a que algunos componentes se montan usando una técnica llamada *through hole* (a través del orificio), es decir, que las patitas de los componentes deben atravesar el circuito por estos agujeros, el uso de esta técnica de montaje depende de la destreza del soldador, de si se usan métodos mecanizados de montaje y de si el elemento a montar soporta este tipo de ensamblado (existen componentes que solamente pueden montarse mediante *through hole*). Para mantener unidos los componentes a la tarjeta, se utiliza soldadura, la cual ofrece unión mecánica (no se desprenderán los componentes) y eléctrica (mantendrá la conductividad eléctrica entre el componente y la pista). La soldadura se solía realizar a nivel industrial con estaño y plomo aunque actualmente se está optando por otro tipo de compuestos para evitar el uso de plomo. Para uso particular, la opción de estaño/plomo es la más comúnmente utilizada.

En este libro nos vamos a centrar principalmente en el punto de creación de prototipos que es donde entra en juego Arduino ayudándonos de manera especial, permitiendo crear de una forma rápida y mediante ayuda de otros componentes prototipos que más adelante se podrían producir de manera industrial.

A lo largo de este libro se irá profundizando progresivamente en el conocimiento de la realización de prototipos con Arduino. Se comenzará desde lo más básico, lo que incluye la instalación del entorno, explicación de la placa Arduino y el funcionamiento de las placas de realización de prototipos (también conocidas *breadboard* o *protoboard*) y se continuará avanzando con conocimiento del lenguaje de programación usado en Arduino, explicando sus características y uso básico, puesto que a medida transcurran los capítulos se aprenderán nuevos conceptos del lenguaje a través de diferentes ejemplos. Una vez que se han obtenido los pilares para la realización de prototipos, se irán creando múltiples ejemplos con código comentado a fin de ir viendo la utilización de distintos componentes y sensores, hasta llegar a ser capaces de realizar proyectos desde cero e incluso crear diseños para producción automatizada.

Es posible que el lector se pregunte si es necesario conocimientos previos de electrónica o programación. La respuesta es no, pero si se tiene conocimiento de electricidad, electrónica o lenguaje de programación C (o alguno similar), le será útil para avanzar más rápido.
En cuanto a los materiales necesarios, lo esencial es un ordenador (con Linux, OSX o Windows) y una tarjeta Arduino; en los ejemplos se usará una Arduino Uno, pero si se tiene otro tipo de tarjeta los montajes son fácilmente adaptables. Además para los ejemplos se necesitarán diferentes componentes como pueden ser diodos, resistencias... sensores como termómetros, detectores de luz y otros tipos de elementos como *shields* (ya se verá qué son), emisores *bluetooth* o pantallas entre otros.
No perdamos más tiempo y comencemos pues con el aprendizaje de la realización de prototipos con Arduino.

Cómo usar este libro

Organización del libro

Este libro pretende ser una guía de iniciación al uso de placas Arduino para realizar prototipos electrónicos. Durante los capítulos posteriores, se irán introduciendo distintos conceptos relativos a la propia placa Arduino, a los componentes que se pueden montan a su alrededor para darle funcionalidad y a la programación necesaria para hacer del conjunto un elemento funcional. Durante los ejercicios se irán introduciendo componentes electrónicos de los cuales se mostrará tanto su funcionamiento como su utilidad, sin entrar mucho en profundidad pero si 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.

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).

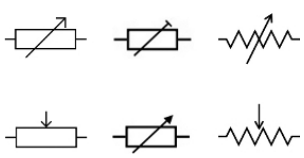


Figura 1. Ejemplos de representaciones de una resistencia variable.

Los capítulos se distribuyen de modo que el lector pueda comenzar de manera rápida a realizar prototipos funcionales casi desde el primer momento. Los primeros capítulos presentan la manera de trabajar con Arduino de modo genérico y su programación básica; una vez conseguidos los fundamentos de utilización, se pasará a conocer distintos componentes y su uso en Arduino; la distribución de los capítulos referentes a elementos electrónicos para utilizar con Arduino se ha hecho de modo que se han empaquetado en unidades por afinidad de comportamiento o utilización, así por ejemplo podemos encontrar una unidad donde se habla de entradas de datos, otra donde el tema son las salidas sonoras, etc...

Es interesante avanzar de modo secuencial a lo largo del libro ya que en los capítulos, se va haciendo uso de lo explicado en capítulos anteriores, pero puede utilizarse también como un libro de soluciones para aplicar a nuestros

montajes, por ejemplo si necesitamos saber cómo funcionan las pantallas de 7 segmentos, podemos dirigirnos al índice e ir directamente al circuito donde se explica a modo de receta.
Este libro no está dirigido a electrónicos ni gurús de la electricidad o programación, sino trata de llevarnos al conocimiento de la creación de prototipos partiendo de cero. Claro está que conocimientos de programación o electricidad harán que avancemos más rápido, pero podemos utilizar el libro si no los tenemos. Junto con la lectura de los capítulos se incorporan pequeños montajes o ejercicios para poner a prueba lo leído; podemos no realizar dichos ejercicios, pero aconsejo realizarlos para afianzar conocimiento y familiarizarnos con la placa.
Aunque a decir verdad... creo que la mejor manera de utilizar este libro es precisamente pensando en qué nuevas cosas se pueden hacer con lo que se explica en él, perfeccionando los montajes, juntando varios ejercicios para conseguir circuitos más complejos, añadiendo nuevas funcionalidades... mejorando lo aquí presente.

Convenios empleados

- Las combinaciones de teclas que en la pantalla aparecen relacionadas con el signo más, como por ejemplo **Ctrl+U**, en este libro aparecen relacionadas con un guión e impresas en negrita, por ejemplo, **Control-U**.
- Los nombres de botones, herramientas así como las combinaciones de teclas aparecen en negrita para facilitar su identificación; por ejemplo, el botón **Guardar**.
- Los nombres de cuadros de diálogo, menús, submenús y fichas aparecen en otro tipo de letra para facilitar su identificación, por ejemplo, el menú Ventana.
- Los comandos consecutivos para seleccionar aparecen separados por el signo mayor que (>) y en el orden de la selección. Por ejemplo, Archivo> Guardar como.
- Elementos como funciones, palabras clave del lenguaje, comandos de programación y en general todo lo relativo a código aparecerá destacado con un tipo de letra `Courier`.
- En los números fraccionarios se usará el punto en lugar de la coma para la separación de los números decimales; manteniendo así coherencia la notación utilizada en Arduino.

En el libro aparecen resaltados una serie de temas o acontecimientos extraordinarios de la siguiente forma:

Nota:

Comentarios o noticias fuera de texto.

Advertencia:

Información importante a tener en cuenta para la integridad del trabajo o del sistema.

Truco:

Consejo o información que puede facilitar un trabajo.

1 Introducción a Arduino

En este capítulo aprenderá a:

- Conocer la familia Arduino.
- Utilizar el entorno de programación.
- Manejar la placa de prototipo breadboard.

¿Qué es Arduino?

De manera formal se podría decir que Arduino es una plataforma de realización de prototipos electrónicos compuesta tanto de hardware como de software. En la parte correspondiente al hardware, Arduino dispone de una serie de tarjetas programables compuestas básicamente por un microcontrolador (basado en los ATMEGA8, ATMEGA168, ATmega328 de Atmel o familias de éstos), un cristal oscilador y un regulador lineal de 5 voltios y que ofrece un cierto número de entradas y salidas tanto analógicas como digitales (pines de entrada y salida). La función del oscilador es proveer al microcontrolador de una serie de pulsos que permiten que pueda funcionar a una determinada velocidad. Dependiendo de los modelos de Arduino tendrán unas u otras entradas y salidas adicionales, por ejemplo la de USB. En general las placas Arduino utilizan microcontroladores de 8 bits excepto la Arduino Due que es de 32 bits.

Nota:

En electrónica se distinguen los microcontroladores de los microprocesadores en que los microcontroladores suelen estar pensados para tareas específicas mientras que los microprocesadores son de uso más genérico. Otra diferencia estriba en que los microprocesadores necesitan de elementos externos para proporcionar entradas y salidas y gestión de memoria, mientras que los microcontroladores tienen embebidas estas funcionalidades. Ejemplos de microprocesadores son los Motorola 68000 o los Intel Pentium, mientras que de microcontroladores tenemos los Atmel AVR o los Texas Instruments TI MSP430.

Si queremos explicar de una manera más informal, definiríamos Arduino como un pequeño ordenador al que se le puede programar para que interactúe con el mundo real, bien ofreciendo salidas o reaccionando a una serie de entradas. Tanto las entradas que puede entender la tarjeta como las salidas que ofrece son de naturaleza eléctrica, dicho de otro modo, el hardware Arduino sólo entiende electricidad, así 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 un actuador como por ejemplo una válvula, adecuar la salida eléctrica a lo esperado por ella).

Para poder programarlo, Arduino utiliza un lenguaje propio llamado *Arduino programming language* que se basa en el lenguaje de programación *Wired* (más información sobre este lenguaje en `http://wiring.org.co/`). Para facilitar su codificación, se proporciona un entorno de trabajo basado en Processing que es un lenguaje de programación de código abierto inicialmente concebido para el aprendizaje y actualmente utilizado para múltiples propósitos desde simulaciones hasta diseño 3D con interactuación (más información en `http://www.processing.org/`) y que más adelante utilizaremos en uno de los ejemplos porque además para gente que no sabe programar es muy rápido de aprender.

Para obtener información del mundo exterior, Arduino utiliza sus entradas de modo que reciben impulsos eléctricos, mediante diferentes elementos se capta la realidad para transformarlo en corriente eléctrica como por pueden ser acelerómetros, detectores de ultrasonidos, botones... es lo que se comúnmente se llaman sensores. Por ejemplo podríamos hacer un detector de proximidad mediante un botón que cuando algo esté cerca lo apriete... pero quizá es demasiado cerca... mejor usar un detector de ultrasonidos ¿no?. En cada situación deberemos seleccionar el sensor que más se adapte a las necesidades.

De la misma forma 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, son lo que se suelen denominar actuadores. Por ejemplo, podríamos crear un semáforo, para lo cual utilizaríamos 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 están pensados para ser utilizados por personas sin preparación específicos de programación ni electrónicos... bueno, un poco si se necesita, pero no es necesario tener grandes conocimientos para realizar montajes realmente interesantes y útiles, y así lo iremos descubriendo a lo largo del libro y sus ejemplos.

Existen multitud de modelos de Arduino, desde las más potentes como la Arduino Mega, con 54 puertos digitales de entrada y salida y 16 entradas analógicas, hasta pequeñas piezas tales como Arduino mini 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

electrónicos o la Esplora con múltiples sensores (micrófono, pulsadores, acelerómetro, sensor de temperatura) y actuadores (altavoz, leds...) ya incluidos en la placa, lo que la hace ideal para aquellos que quieren utilizar rápidamente una placa sin tener conocimientos de electrónica o para controlar algún otro dispositivo. Para el desarrollo de periféricos destinados a la plataforma de movilidad Android, existe la Arduino Mega ADK (Accessory Development Kit), ya preparada para conectividad con dispositivos móviles con sistema operativo Android.

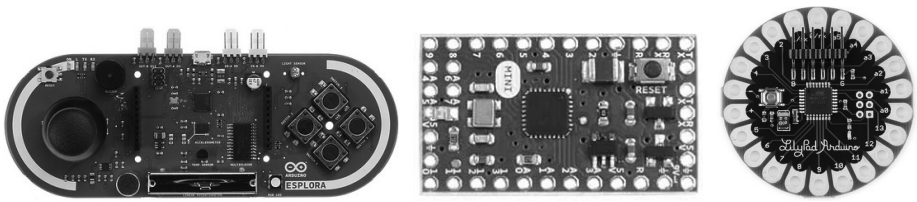


Figura 1.1. Placas Arduino Esplora, Mini y LiliPad.

Además también hay múltiples tarjetas compatibles realizadas por la comunidad como por ejemplo la Freeduino o la Roboduino y es que tanto los esquemas del hardware Arduino como el código fuente de su software son libres, cualquiera puede acceder a ellos y crear sus propias placas y venderlas. Además todas estas placas copias de la Arduino, normalmente mantienen la disposición de los pines de modo que cualquier complemento existente para la original como los *shields* (unas líneas más adelante se explica que son) pueda funcionar con ellas. Pese a que como se ha visto existen varias placas Arduino, nos centraremos en una de las más extendidas, concretamente la Arduino Uno (muy semejante a la Arduino Duemilanove) aunque si el lector tiene otra diferente, con muy pocos cambios (como cambiar el número de pin donde realizar las conexiones) podrá adaptar los ejemplos. La última revisión 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).

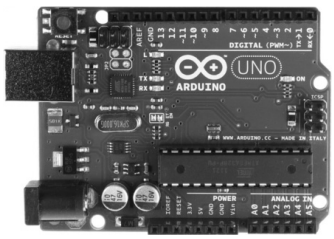


Figura 1.2. Placa Arduino Uno R3.

Shields

Para facilitar los montajes con piezas requieran conexiones complejas o múltiples (como por ejemplo pantallas TFT), los fabricantes ponen a disposición de Arduino unos bloques de conexiones ya ensambladas a modo de placas que se montan directamente sobre la tarjeta Arduino, siendo totalmente operables nada más ser colocadas. Estos bloques son conocidos como *shields* (escudos) y existen de muchos tipos (Ehternet, GPS...).

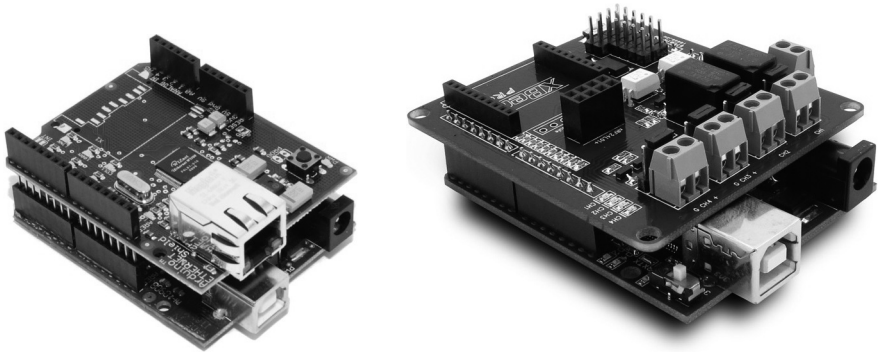


Figura 1.3. Placas Arduino con shield montado.

Normalmente las tarjetas *shield* llevan emparejada una funcionalidad, es decir en la propia placa *shield* van embebidos los componentes que ofrecen la funcionalidad, como por ejemplo la Ethernet lleva el adaptador para la clavija RJ45 o la de comunicación GSM lleva el habitáculo para introducir la tarjeta 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 TFT existen adaptadores que facilitan su ensamblado.
El uso de los *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. En el ejemplo de ensamblado de la pantalla TFT, el adaptador nos configura directamente los 40 pines de conexión de la pantalla, sin tener que preocuparnos de conocer la función de cada uno de ellos. Véase la figura 1.4.
En caso de usar *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 apiladas, alimentando los pines necesarios de cada una de ellas. Véase la figura 1.5.

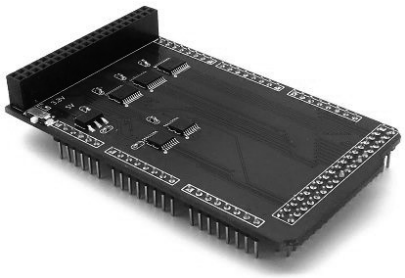


Figura 1.4. Placa *shield* de adaptación para TFT.

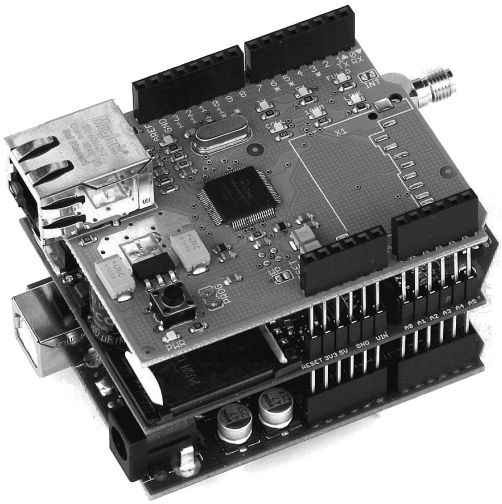


Figura 1.5. Placa Arduino con dos *shield* montados.

Arduino Uno

Como se ha comentado anteriormente, el modelo de placa Arduino a utilizar como referencia en el libro es la Arduino Uno por ser una de las más difundidas (ella y sus clones). Su aspecto ya se pudo ver en una figura anterior (y posiblemente lo esté viendo en vivo en sus manos), pero vamos a ver que es cada una de las partes de la placa.

1. Pulsador de Reset: Sirve para inicializar de nuevo el programa cargado en microcontrolador.
2. Puertos de entrada y salida digital: Son los zócalos donde conectar los sensores y actuadores que necesiten señal digital.

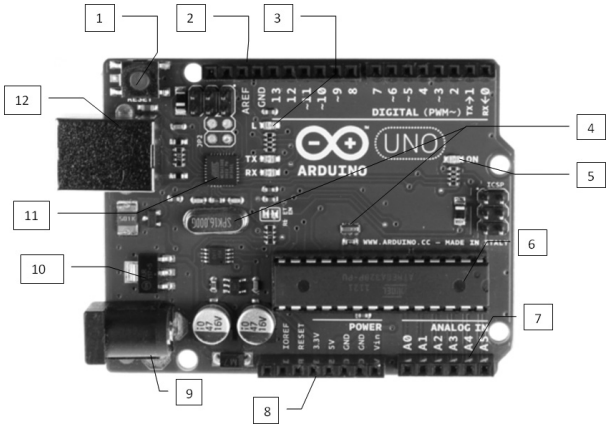


Figura 1.6. Elementos de la placa Arduino Uno.

3. Led pin 13: Es un led integrado en la placa para poder ser utilizado en montajes. Corresponde al pin 13.
4. Reloj oscilador: Es el elemento oscilador que permite mantener la frecuencia de funcionamiento del microcontrolador. En el caso de Arduino Uno, la frecuencia de funcionamiento es de 16 Mhz.
5. Led de encendido: Este se ilumina cuando la placa está correctamente alimentada.
6. Microcontrolador: El cerebro de la placa. Actualmente se montan con un ATmega328 de Atmel, pero se puede encontrar con microprocesadores ATmega8 o ATmega168, en tal caso las conexiones son idénticas. En cuanto a memoria consta de 32 Kb de memoria Flash, 2k de SRAM y 1K de EEPROM. También dispone integrado un transmisor/receptor asíncrono universal (*UART Universal Asynchronous Receiver/Transmitter*) que se utilizará en comunicaciones serie.
7. Entradas analógicas: Zócalo con distintos pines de entrada analógica que permiten leer entradas distintas de los 0 y 1 digitales.
8. Zócalo de tensión: Aquí encontraremos pines con los cuales alimentar nuestro circuito y entradas de referencia voltaica.
9. Puerto de corriente continua: Se utiliza para alimentar la placa si no se usa alimentación vía USB.
10. Regulador de tensión: Sirve para controlar la tensión a enviar a los terminales de alimentación. La placa puede ser alimentada tanto por USB como por alimentación externa. La alimentación externa puede ser de 6 a 20 voltios, mientras que el suministro de la placa es de 3,3 y 5 voltios.

En caso de ser la alimentación menor de 7 es posible que la salida de 5 voltios ofrezca menos de este valor (lo que se debe tener en cuenta en algunos montajes), mientras que si se alimenta con más de 12 voltios la placa puede sobrecalentarse. El regulador es el encargado de controlar que las salidas internas tengan la tensión deseada de 3.3 y 5 voltios.

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.

    Si se utiliza la placa Arduino Uno no hay que tener especial precaución a la hora de alimentarla con tensión (siempre que se respeten los límites de 6-20 voltios) dependiendo de si se usa USB o alimentación externa, ya que es capaz de detectar la fuente de alimentación y conmutar automáticamente entre USB y alimentación externa, pero no es así en todas las placas, existen algunas como la Arduino Diecimila o clónicas como la Roboduino que no son capaces de detectarlo y en lugar de hacerlo de modo automático hay que modificar la configuración física de la placa bien cambiando de posición unos *switches* (interruptores) o bien modificando algún jumper (piececitas de plástico que sirven para conectar haciendo puente entre distintos pines macho).

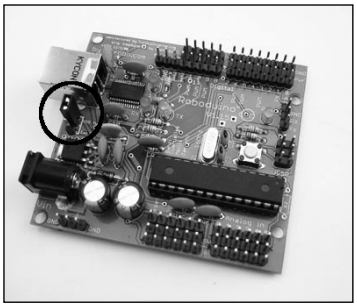


Figura 1.7. Jumpers de selección de alimentación en Placa Roboduino.

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 integrado (IDE) 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 1.0.5, siendo obligatorio el uso de la versión 1.5.2 beta (en estado de prueba) para las placas Due (que trabajan en 32 bits en lugar de en 8 bits como el resto).

Tras la descarga correspondiente se obtiene un fichero con el nombre arduino-<versión>-<plataforma>.zip. 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 obtiene la estructura de directorios siguiente:

- drivers: Directorio con los drivers para poder comunicarse con las placas desde el entorno de desarrollo y poder así transferir el programa a la tarjeta entre otras funcionalidades. Cada plataforma tiene unos drivers específicos, que vienen en su fichero correspondiente.
- examples: Aquí encontraremos una serie de ejemplos que se pueden leer desde el entorno de programación, con código para controlar varios tipos sensores, manejo de cadenas, etc...
- hardware: Contiene herramientas e información acerca del hardware Arduino para uso interno del entorno de desarrollo.
- java: En este directorio se encuentra un entorno de ejecución Java (JRE) para poder utilizar el entorno de programación.
- lib: Aquí podemos encontrar librerías Java así como los gráficos del tema del entorno de desarrollo, los gráficos que configuran su aspecto; por lo que si queremos hacer nuestro propio tema, podemos entonces variarlo desde aquí.
- libraries: Son librerías correspondientes a Arduino y su programación, existen ciertos componentes cuya complejidad de programación se ve drásticamente reducida al usar librerías de ayuda. Aquí encontraremos librerías para varios de estos componentes como por ejemplo para escribir en una SD.
- reference: Documentación sobre Arduino.
- tools: Conjunto de herramientas para poder llamar directamente desde el entorno de desarrollo Arduino.

Antes de poder trabajar con la tarjeta Arduino, y dependiendo del sistema operativo que tengamos, puede ser necesario instalar ciertos drivers. Los drivers son dependientes de la plataforma sobre la que se vaya a trabajar, es decir no se pueden instalar los drivers de Windows en Linux. Según el sistema operativo que se vaya a usar, 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 de Mac y Linux son muy semejantes y básicamente es descomprimir el fichero y ponerse a trabajar, ya que la configuración estándar del núcleo de estos sistemas normalmente permite trabajar con las tarjetas Uno y Mega sin realizar ningún ajuste más, en caso de tener un sistema antiguo, kernel compilado a media o tarjetas diferentes habría que leer las instrucciones de instalación.
En caso de utilizar Windows el proceso es un poco más largo; el fichero tendrá un nombre semejante a arduino-1.0.5-windows.zip. Tras descomprimirlo hay que conectar la placa Arduino al ordenador y saltará un mensaje conforme no se tienen instalados los drivers para el elemento USB conectado, intentará buscarlos e instalarlos pero fallará.

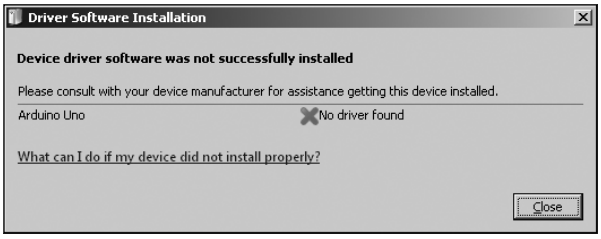


Figura 1.8. Error al instalar los drivers Arduino Uno.

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. Véase la figura 1.9.
Si se pulsa mediante el botón derecho en 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. Véase la figura 1.10.

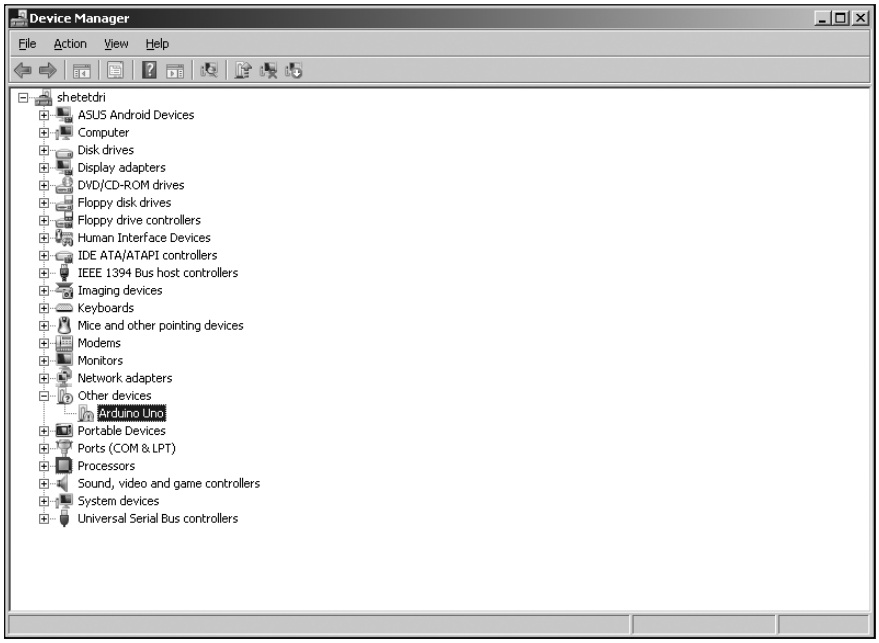


Figura 1.9. Error en el Gestor de Dispositivos.

Figura 1.10. Actualización del driver para Arduino Uno.

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

Figura 1.11. Driver encontrado, se procede a la instalación.

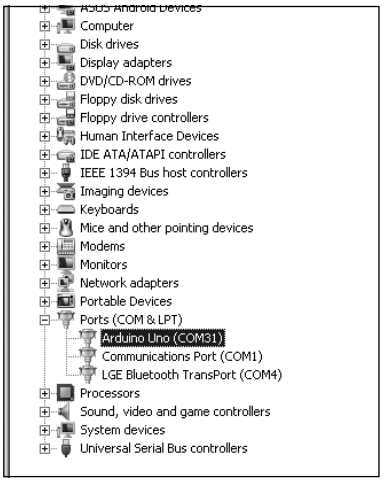


Figura 1.12. Instalación del driver completada.

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`.

Entorno de programación

Para ejecutar el entorno de programación habrá que seleccionar el fichero **arduino.exe** (o **arduino** en Linux y Mac) dentro del directorio de donde se ha descomprimido el fichero `.zip`. El aspecto del entorno de desarrollo es el de la figura 1.13, aunque dependiendo del sistema operativo que se esté utilizando puede haber ligeras diferencias, el aspecto general el muy similar.

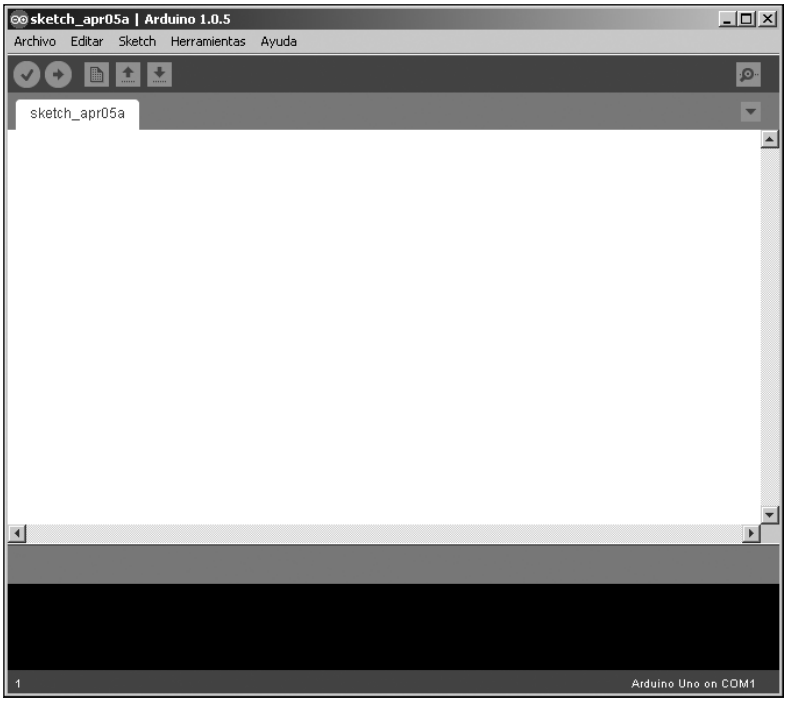


Figura 1.13. Entorno de desarrollo de Arduino.

Se puede observar que la ventana del entorno de desarrollo está dividida en tres partes 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 se encontrará el código fuente también llamado *sketch*, mientras que la parte inferior corresponde a la salida de la consola, donde se podrán ver errores y mensajes de información durante el proceso de codificación. Debajo de esta consola se encuentra una barra de información donde se puede encontrar a la izquierda 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 en el que se encuentre conectada.
Dependiendo de la versión del entorno, la botonera puede tener un distinto número de botones, en este caso, la versión es la 1.0.5 y entonces existen seis botones.

Figura 1.14. Botonera del entorno de desarrollo de Arduino 1.0.5.

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: Verifica la sintaxis del código fuente que esté cargado en dicho momento.
- Cargar: Transmite el programa a la placa Arduino que esté conectada al ordenador en ese momento; 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: 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 `.ino` (en versiones anteriores la extensión era `.pde`, pero pueden abrirse sin problemas en las versiones actuales del entorno ya que realmente son archivos de texto plano). Se guardará un archivo `.ino` por cada pestaña de código que se tenga abierta.
- Monitor serie: Por último y situado a la derecha del todo, se encuentra el botón del monitor serie. Este monitor es una herramienta muy importante sobre todo cuando se trata de depurar el programa. El monitor muestra los datos enviados por el puerto serie o USB desde Arduino y también permite el envío de datos hacia la placa. Pulsando sobre el botón se abre una nueva ventana mostrada en la figura 1.15.

Figura 1.15. Monitor serie.

En el monitor serie se puede observar un primer campo en la parte superior; dicho campo permite introducir datos para enviar hacia la placa Arduino, por ejemplo emular la entrada de datos de un terminal mientras no se tenga el teclado físico que iría en el montaje final; 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 selectores, el primero de ellos permite controlar el comportamiento de nueva línea y retorno de carro en los mensajes y el segundo se encarga de configurar la velocidad de comunicación con la placa Arduino medido en baudios (*baud*), el *baud* es el numero de cambios del estado de los bits 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 depurar programas como para introducir datos hacia la placa Arduino.
En cuanto a la zona de menús vamos a ver las entradas más importantes. Dentro del menú Archivo podemos encontrar las siguientes entradas:

- Nuevo: Permite crear un nuevo *sketch* en blanco.
- Abrir: Sirve para abrir *sketches* ya existentes. Genera una nueva ventana con el *sketch* seleccionado.
- Sketchbook: Es la librería de *sketches* guardados.

- Ejemplos: Contiene una serie de ejemplos de *sketches* operativos que podemos utilizar en nuestros trabajos. Se presentan convenientemente ordenados por tipo.
- Cerrar: Cierra el *sketch* actual.
- Guardar: Guarda el *sketch* activo.
- Cargar: Carga el *sketch* actual en la tarjeta configurada que esté conectada en el puerto configurado.
- Cargar usando programador: Carga el sketch en el microcontrolador utilizando un programador en lugar de la tarjeta Arduino. Esta opción no se usará en el libro.
- Preferencias: Nos guía a la pantalla de preferencias donde configurar el comportamiento del editor. Más adelante se explica en detalle.

En el menú Editar encontramos:

- Copiar, Pegar...: Dan acceso a funcionalidades típicas de cualquier editor de textos.
- Copiar como HTML: Copia el código del *sketch* en formato HTML para poderse insertar en páginas Web.
- Copiar para el Foro: Copia el código del *sketch* en un formato especial para usar 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: Permite comentar y descomentar bloques de código.
- Buscar: 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: Procede a verificar la sintaxis del programa, es idéntico a la acción realizada por el botón comentado anteriormente.
- Mostrar la carpeta de Sketch: Abre el explorador de sistema en la carpeta que contiene el *sketch* actual.
- Agregar Archivo: Permite añadir nuevos archivos de código al *sketch* activo en ese momento.
- Importar Librería: 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 encontramos:

- Formato automático: Organiza y ordena el código de modo que quede perfectamente tabulado y fácil de leer.
- Archivar el Sketch: Crea un fichero comprimido con todos los códigos fuente necesarios correspondientes al *sketch*. Ideal para portar los *sketches* entre distintas máquinas o realizar backups.
- Monitor serie: Muestra el monitor serie del mismo modo que el botón comentado anteriormente.
- Tarjeta: Muestra una serie de tarjetas entre las cuales tenemos que seleccionar la que se quiere utilizar para probar el *sketch*.
- Puerto serie: Muestra los puertos serie disponibles. Se debe seleccionar aquél en el que se encuentre conectada la tarjeta sobre la que se quiere trabajar.
- Programador: 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: 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 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.
Existen otros valores que pueden modificarse directamente sobre el fichero `preferences.txt` dentro del directorio de instalación del entorno y pudiendo accederse a él mediante el enlace proporcionado en la parte inferior de la ventana de preferencias. En caso de modificar manualmente este fichero es muy recomendable hacerlo cuando el entorno de programación no se esté ejecutando y hacer previamente una copia de seguridad.

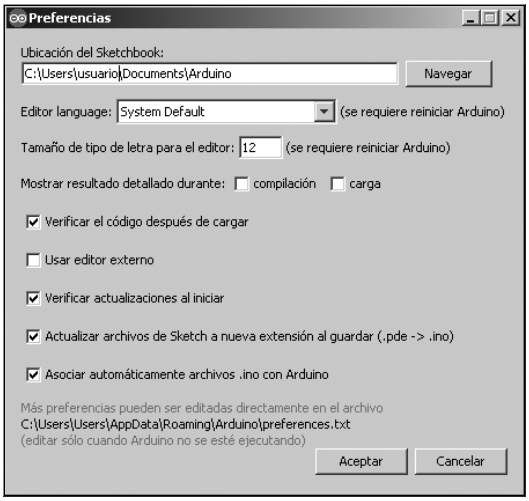


Figura 1.16. Ventana de preferencias.

En la parte central es donde codificaremos los programas o *sketches*. El entorno de Arduino no ofrece tantas ayudas como otros entornos de programación al uso (como lo hace Eclipse con Java por ejemplo), lo que sí hace es ofrecer colores en el texto para la sintaxis de programación pero aún no tiene nada de autocompletados ni funciones avanzadas, aunque seguramente las veamos ir añadiendo a lo largo de las próximas versiones.

Protoboard o breadboard

La placa Arduino está pensada para interactuar con el mundo real y esta interactuación la realiza través de sensores para obtener información del mundo exterior, por ejemplo tomar 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 (como puede ser la Esplora), 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 1.17. 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 componentes y extremos de los cables para simular el circuito impreso. Ciertos agujeros se hayan interconectados entre sí directamente en la placa y permiten crear conexiones con menos elementos.

Puesta en horizontal, el protoboard tiene uno o dos buses de alimentación en los bordes superior e inferior, normalmente marcados con los signos + y - para notar las polaridades 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 se da tensión en uno de ellos, esta existirá en todos los agujeros de la línea. En los ejemplos del libro se utilizará el polo positivo para como línea de alimentación y el negativo como línea de tierra. Véase la figura 1.18.

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 manera que todos los

agujeros correspondientes a un bloque y de modo vertical, serían el mismo nudo de un circuito, o dicho de otro modo tendrían la misma tensión. Los bloques están separados entre sí eléctricamente. Véase la figura 1.19.

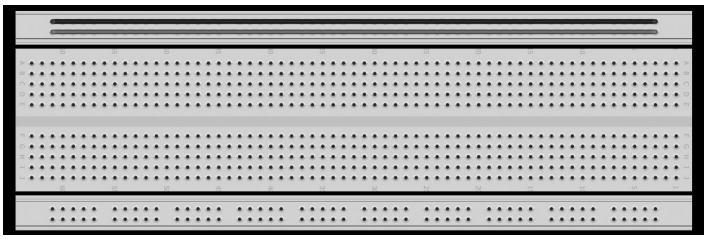


Figura 1.18. Protoboard con dos zonas de alimentación.

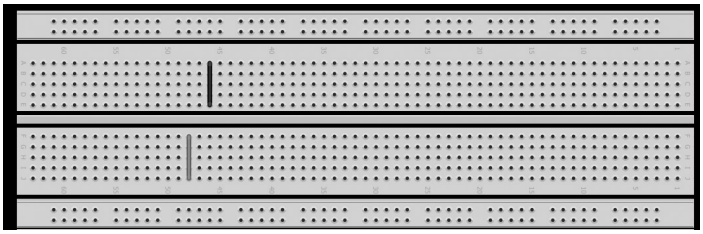


Figura 1.19. Zona de montaje en la protoboard.

Las protoboard suelen venir acompañadas de unas letras y números de modo que cada orificio de la placa puede identificarse por la dupla 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.

Primer Sketch

Antes de adentrarnos en la programación y como supongo que ya hay ganas de probar alguna cosa después de tanta lectura, vamos a realizar ahora el primer proyecto, pero será tan sencillo que ni necesitaremos la protoboard, sino que utilizaremos el led (lucecita) que viene integrada en la propia placa Arduino Uno.

Vamos a utilizar un ejemplo disponible dentro del propio entorno de desarrollo. Para ejecutarlo lo primero que debemos hacer es abrir el entorno de desarrollo y mediante el menú Archivo>Ejemplos>1.Basics>Blink abrir el *sketch* para ver su código fuente.
Aún no vamos a entrar a analizar el código, en este caso simplemente vamos a cargarlo en la tarjeta y ejecutarlo.

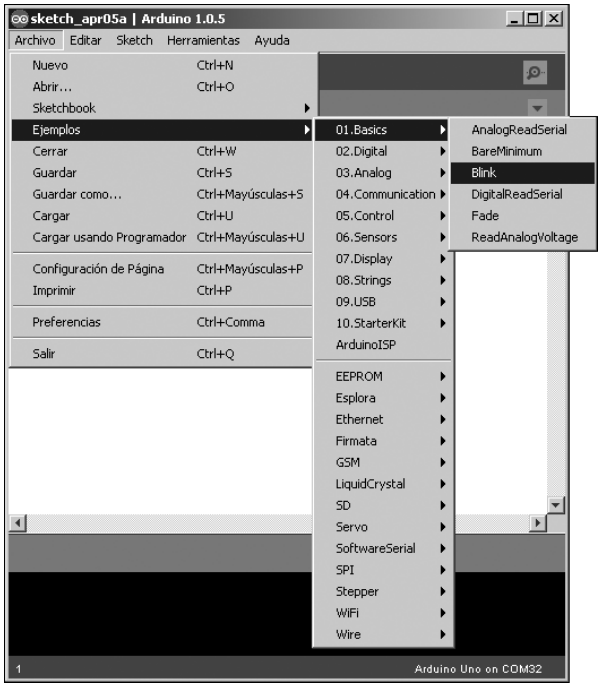


Figura 1.20. Selección del *sketch* de ejemplo.

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 convenientemente alimentada. Antes de poder ejecutar el *sketch* hay que asegurarse que se tiene seleccionada correctamente la tarjeta a utilizar mediante el menú Herramientas>Tarjeta.
De la misma manera se debe seleccionar el puerto al que está conectada la tarjeta mediante Herramientas>Puerto Serial.
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.

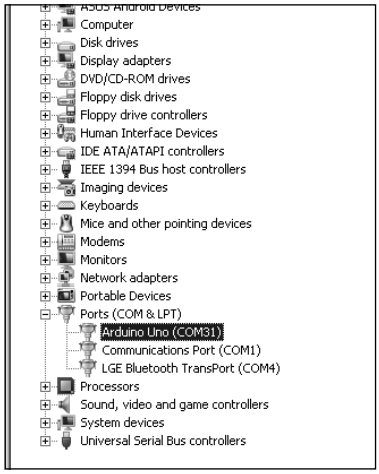


Figura 1.21. Tarjeta Arduino Uno conectada en el puerto serie COM31.

Pulsando el botón  o mediante el menú Archivo>Cargar, se procede al envío del programa al microcontrolador. 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 serigrafía 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 continua. 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.

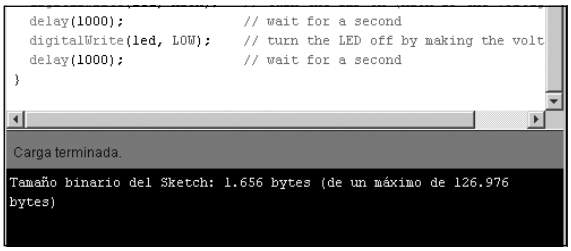


Figura 1.22. Mensaje de carga completa en el entorno de desarrollo.

2 Lenguaje de programación Arduino

En este capítulo aprenderá a:

- Conocer los fundamentos de la programación.
- Usar el entorno de desarrollo.
- Crear *sketches* desde cero.
- Usa el monitor serial.

Si se ha comprado una placa Arduino, es porque se le quiere dar una utilidad o probar su funcionamiento, dicho de otro modo, queremos hacer algo con ella. Para poder indicar qué debe hacer y bajo qué estímulos o condiciones, utilizaremos la programación.
La programación de Arduino está basada en C, por lo que aquellos que tengan una base en C o Java (por usar estructuras semejantes) les será mucho más sencillo avanzar en este capítulo, pero como se verá es muy sencillo crear programas.

General

Los *sketches* de Arduino están compuestos ficheros que contienen las instrucciones a ejecutar por el microcontrolador. Estos ficheros a su vez se componen de bloques de código con una finalidad determinada. Los bloques de código se encuentran entre llaves que sirven para marcar el alcance de cada uno de ellos. Estos bloques pueden identificar funciones, bucles, condiciones...
A diferencia de otros lenguajes de programación, aquí no se debe terminar cada línea de código con algún carácter, pero si se debe terminar cada instrucción mediante el carácter ";" (no confundir línea de código con instrucción... normalmente coinciden, pero no tiene porqué), es decir podemos dar una instrucción y varias líneas más abajo poner el ";" siempre que no haya más instrucciones por el medio.

Aspecto

Todos los programas de Arduino tienen un aspecto similar en su composición principal, del mismo modo que en C podemos encontrar el `main()`, en Arduino los programas son semejantes a:

```
// Inclusión de librerías

// Definición de variables;

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

// Bucle principal
void loop() {
   // se ejecuta periódicamente
}
```

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

Lo primero de todo es comenzar importando las librerías necesarias (es posible que no se necesite ninguna). Las librerías son bloques de código creados por nosotros mismos o por terceros, que encapsulan instrucciones con funcionalidad establecida, y que pueden ser reutilizados en otros programas. Evitan que se reinvente la rueda.
Tras la importación de las librerías, se definen las variables globales que se van a utilizar en el programa. Las variables nos servirán para mantener datos en memoria y las definidas en este área 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).
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. Se ejecuta una y sólo una vez por cada encendido de la placa y nos sirve para configurar el uso de pines, inicializar variables...
El bloque `loop()` es el corazón del *sketch*, como su propio nombre indica ejecuta una y otra vez en bucle lo que se encuentre en su interior. Se ejecuta después del bloque `setup()` y es el que tiene el control de la placa Arduino. Será dentro de él donde tendremos que leer las entradas, procesar los datos y escribir en las salidas.
Por último el bloque `function()` en este caso nos sirve para representar 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.
Realmente las únicas secciones necesarias para que compile el *sketch* son los bloques `loop()` y `setup()`, si bien casi con toda seguridad necesitaremos hacer uso de alguno de los otros bloques o de todos ellos.

Estilo

El lenguaje de programación de Arduino, permite un estilo de programación bastante laxo, siempre que se cumplan unas pocas reglas claro. Algunos lenguajes de programación exigen una codificación manteniendo unas tabulaciones entre líneas, obligan a dejar ciertos espacios entre operandos o no permiten tener más de una instrucción por línea. Nada de esto aplica a Arduino.

Las instrucciones siguientes sirven para definir tres constantes:

```
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.
Las instrucciones en Arduino vienen marcadas por el carácter ";" que marca fin de instrucción y comienzo de una nueva, y si bien es cierto que se suele poner una instrucción por línea de código para facilitar su lectura, nada impide poner varias instrucciones por línea.
Los bloques de código vienen dados por la pareja de llaves que encierran el código correspondiente, pero estas llaves pueden ser colocadas a gusto del programador. Normalmente se usan dos estilos, poniendo la llave de apertura en la misma línea que el identificador de bloque:

```
void function(){
//datos
}
```

O poniéndolo en una línea inferior para marcar más claramente el bloque:

```
void function()
{
   //datos
}
```

En el libro se utilizará la primera notación pero son igualmente válidas.
Respecto al sangrado o tabulación de las líneas no existe tampoco una restricción, existen lenguajes donde los bloques de código vienen marcados por la tabulación de las instrucciones, pero en este caso se utilizan las llaves, por lo que podemos utilizar la tabulación que más nos apetezca. También es verdad que es muy recomendable ser ordenado en las tabulaciones de las instrucciones, ya que más adelante permite leer mejor el código y encontrar rápidamente a que bloque pertenece cada instrucción. El entorno de programación de Arduino proporciona una utilidad para ordenar el código y tabularlo de manera precisa, se realiza mediante el menú Herramientas>Formato Automático o mediante la combinación de teclas **Control-T**.

Comentarios

Los comentarios son anotaciones que se hacen en el código con tal de ser más fácil entenderlo más adelante. A medida que los programas son más complejos, se hace más difícil entender el código escrito y para facilitar esta tarea se deben añadir comentarios en lenguaje humano. Los comentarios no afectan al código ni al tamaño final del programa a ejecutar en el microcontrolador (aunque lógicamente si al tamaño del fichero de código fuente). Dentro de Arduino, los comentarios pueden ser de dos formas, de línea y de bloque.

El comentario de línea sirve para hacer pequeñas anotaciones de una sola línea y suele utilizarse para aclarar las inicializaciones de variables y situaciones semejantes.

Para realizar un comentario de este estilo, se introduce una doble barra que marca comienzo de comentario; el final de comentario lo marca el fin de línea. Por ejemplo:

```
const int led1Pin =  3;      // Es el led verde
const int led2Pin =  4;      // Es el led rojo
// Para hacer comentarios de varias líneas
// hay que poner las barras en cada una de ellas
```

El comentario de bloque permite más flexibilidad en su uso. Para comenzar el bloque comentado se introducen los caracteres "/*" y para terminar el bloque "*/". Todo lo que se encuentre entre medias se considera comentario. La primera diferencia con respecto al método anterior es que podemos introducir más de una línea de manera sencilla.

```
/* Definimos los pines
para los leds
pin 3 -> led verde
pin 4 -> led rojo */
const int led1Pin =  3;
const int led2Pin =  4;
```

La segunda diferencia es que mientras que el código de línea se pone al final de cada línea (ya que no existe una marca para indicar el final de comentario que no sea el propio final de línea), el de bloque puede ponerse donde se quiera (siempre que nos acordemos de cerrar el bloque) incluso en mitad de la definición de un bloque, aunque es desaconsejable por no permitir una lectura sencilla del código.

```
/* Vamos a definir un bloque */
void function /*Es un bloque de función*/ function{
...
}
```

Los comentarios pueden ir en cualquier idioma puesto que no interfieren en el código, pero el si se usa el español se debe tener en cuenta que tiene caracteres que no son ASCII estándar como pueden ser los acentos (tildes) y la ñ, y éstos pueden hacer que al pasar el código entre distintos sistemas, llegue a corromperse.
En el entorno de programación veremos que los comentarios se diferencian del código, porque se muestran en letra gris en lugar de negra.

Las variables

Las variables son la parte del código que sirve para almacenar datos en memoria y trabajar más adelante con ellos. 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 un carácter numérico o una letra, bien sea mayúscula o minúscula y sin espacios. Algunos ejemplos de nombres de variables son los siguientes: `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 bucles y contadores suelen utilizarse variables de una sola letra.
Cuando se define 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 tiene un alcance de visibilidad (denominado *scope*) determinado por 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 sólo visibles desde dentro de la propia función; más aún, las variables son solamente visibles dentro del bloque en las que se declaran, así si se declara una variable dentro de un bloque de bucle, tan sólo se puede acceder a ella desde el propio bucle y no desde fuera.
Para definir la variable no sólo hace falta el nombre de esta, también hay que indicar el tipo de valor que va a contener la variable, si será una cadena de texto, un carácter, un número entero... Esto sirve para dos cosas:

- Permite al compilador tener una lista de variables con tipos y poder chequear en busca de errores, por ejemplo multiplicar una cadena de texto por un entero.

- 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 que soporta.

Los tipos de variables se representan por una serie de palabras reservadas (que no podemos usar como nombre de variables ni funciones). En Arduino los tipos disponibles son:

- void: Es el tipo nulo. Se usa sólo 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: Mantiene 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 igual que en C, al guardar un carácter se está guardando su código ASCII, lo cual permite realizar operaciones aritméticas (como por ejemplo para pasar a mayúsculas) sobre este tipo de variables. Ej: `char killMe = 'X'; char other = 64 // es la @`
- unsigned char: Mantiene valores de 0 a 255, es lo mismo que el tipo byte y se aconseja el uso de byte en lugar de este tipo de dato. Ej: `unsigned char years = 33;`
- byte: Guarda un valor de 0 a 255. En memoria ocupa (como su nombre indica) 1 byte (8 bits). Ej: `byte years = B100001; //33 en decimal.`
- int: En memoria ocupa 2 bytes (16 bits) y utiliza complemento a dos para guardar los números negativos (más información sobre complemento a dos ver apéndices) lo cual permite guardar valores entre -2^{15} y $(2^{15}) - 1$ es decir entre -32,768 y 32,767. El hecho de trabajar en complemento a dos con números negativos hace que se deba tener especial cuidado en operaciones de *bitshift* o desplazamiento de bits. 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 $(2^{16}) - 1$), es decir de 0 a 65,535. Ej: `unsigned int years = 33;`
- word: Es idéntico en memoria al tipo `unsigned int` y como él, puede guardar valores numéricos desde 0 hasta 65,535. Ej: `word years = 1880;`

- long: Mantiene valores numéricos superiores al tipo `int`. En memoria ocupa 4 bytes (32 bits) y puede guardar valores negativos. El rango de valores es de -2,147,483,648 a 2,147,483,647 o dicho de otro modo de -2^{31} a $(2^{31}) - 1$. Ej: `long amount = -34254329L;`
- unsigned long: Parecido al tipo `long` pero sólo para valores positivos. Ocupa en memoria 4 bytes lo que da lugar a un rango de valores de 0 a 4,294,967,295 que es $2^{32} - 1$. Ej: `unsigned long amount = 34254329L;`
- float: Guarda datos en punto flotante (ver apéndice) con una precisión de 6-7 decimales. Permite guardar en 4 bytes valores desde -3.4028235E+38 a 3.4028235E+38. Son tipos de datos muy lentos de operar por lo que se debe evitar usarlo en bucles con múltiples repeticiones bajo pena de ralentizar el programa. Las operaciones con estos valores pueden no tener resultados exactos por lo que siempre que sea posible se debe trabajar con enteros. Ej: `float pi = 3.1416;`
- double: Debería doblar la precisión del tipo `float` pero por el momento es idéntico a éste.
- array: El array o arreglo es una colección de variables del mismo tipo que se acceden mediante un índice que indica la posición en la que se encuentra el valor dentro de la colección. Cuando se declaran, se debe indicar el tipo del cual es la colección. También se puede indicar el tamaño del array informándolo entre corchetes *[]*. Ej: `int days[] = {22, 11, 19, 12}; long speeds[13];`. El primer ejemplo crea un array de 4 posiciones de tipo `int` y el segundo un array de 13 posiciones de tipo `long`.

  Para acceder a cada valor sería mediante su indicie, por ejemplo para acceder al segundo valor del array `days`, hay que acceder al índice 1 (ya que empieza a contarse en 0 en lugar de en 1). Ej: `int day2 = days[1];`
- string: Si analizamos bien lo que es un array y lo comparamos con una palabra, podemos observar que una palabra no es más una array de caracteres. En Arduino al igual que en C, estos arrays de caracteres que realmente son cadenas de texto tienen una particularidad y es que la última posición del array debe ser un carácter nulo (lo que facilitará el tratamiento de la cadena en el código) por ejemplo una cadena la podemos expresar como cualquiera de estas formas: `string myString[8] = "viernes";` `string myString[8] = {'v','i','e','r','n','e','s','\0'};` `string myString[] = "viernes";` Todas ellas son semejantes. Hay que fijarse que la palabra "viernes" tiene siete letras y el espacio del array es de 8, esto es porque el último valor ha de ser el valor nulo para marcar

el fin de la cadena. El valor nulo viene representado por `\0`. En el último caso donde no se ha dado tamaño al array, el compilador se encargará de calcular su tamaño de modo automático para que quepa la cadena de texto asignada.

- String: lo primero que debe notar en este tipo de datos es que la palabra está en mayúscula y es que denota una clase en lugar de un tipo básico, es decir nos servirá también para representar una cadena de texto pero además esta clase lleva asociada múltiples funciones (que ya descubriremos) para poder trabajar con textos, por ejemplo funciones para obtener la longitud de la cadena o comparar cadenas entre sí.

Advertencia:

Los arrays tienen como índice del primer elemento el 0, es lo que se llama zero indexed *o* zero based, *entonces si se crea un array de 5 posiciones, el índice de la última es 4. Además, en caso de intentar acceder al elemento con índice 5, no daría error como en Java, sino que devolvería lo que hubiera en memoria una posición más allá de donde se encuentre alojado el array, devolviendo realmente cualquier dato totalmente impredecible y en caso de escribir más allá de los límites del array, el resultado puede ser que el programa se vuelva inestable.*

Las variables se pueden declarar allá donde nos interese no hay obligación de hacerlo en un lugar específico como en otros lenguajes; podemos tenerlas al principio de los bloques o justo antes de usarlas, esto queda a decisión del programador para su comodidad.

Modificadores

Cuando se declaran las variables, se les puede añadir ciertas palabras reservadas que modificarán el comportamiento estándar de las mismas. Estos modificadores son:

- static: Al añadir la palabra `static` a una variable se indica que la variable es visible sólo en la función en la que está declarada y que su valor debe persistir entre distintas llamadas a la misma función. Cuando se llama por primera vez a una función con variables estáticas en su interior, éstas se inicializan y a partir de ese momento hasta la finalización de la vida del programa las variables persisten y en las siguientes llamadas a dicha función ya no tendrán entonces que inicializarse de nuevo, sino que

conservarán el valor que tenían durante la llamada anterior. Mantienen el valor de las variables entre distintas llamadas a la función. Ej: `static int years = 10;`

- volatile: Es una directiva de compilador que modifica el modo en el que éste tratará la variable. No tiene nada que ver con el `volatile` de lenguajes como Java. En Arduino indica que no se optimice el uso de la variable y que se cargue desde RAM en vez de hacerlo desde los registros del microprocesador. Se usa cuando se trabaja por ejemplo con interrupciones. Ej: `volatile int error Num = 0;`
- const: Esta convierte la variable en una constante, no se puede modificar su contenido y en caso de intentarlo se recibirá un error de compilación. Ej: `const float gravity= 9.8;`

Constantes enteras

Cuando se programa es habitual utilizar constantes numéricas para asignar a variables o constantes por ejemplo en `int years = 123;` 123 es una constante (no `years`... que es una variable, sino el 123 en sí) ya que siempre tendrá el valor 123. En Arduino estas constantes se tratan como enteros con signo en base 10 por defecto (tipo `int`), pero se puede hacer que se traten de distintas maneras.
Los modificadores para cambiar el tipo son 'U', 'L' y 'UL' (también es posible en minúsculas). Mediante 'U' se especifica que el dato se debe tomar como sin signo. Mediante 'L' se dice que el dato debe ser tomado como un `long`. Por último 'UL' significa que debe ser tomado como un `unsigned long`. Por ejemplo: `unsigned long meters = 9823112ul;`
Para indicar otro tipo de base que no sea la base decimal, Arduino ofrece también algunos modificadores, concretamente disponemos para las bases binaria, octal y hexadecimal (muy útiles cuando se trabaja a nivel bits). Los modificadores son 'B' para binario, '0' para octal y '0x' para hexadecimal y se deben poner delante del valor que se quiere expresar en la base.
El modificador binario sólo puede tener como caracteres 1 y 0. Trabaja en 8 bits dando valores entre 0 y 255. En caso de ser necesario indicar valores binarios de más de 8 bits se debe hacer manualmente multiplicando el byte superior por 255:

```
int myValue= (B10000101 * 256) + B11101100;  //34284 en decimal
```

El modificador octal expresa en base 8 y sólo puede tener como caracteres números del 0 al 7. Ej: `int myValue = 0123; //83 en decimal.`

El modificador hexadecimal expresa el número en base 16 y los caracteres válidos son los números y las letras de la 'A' a la 'F'. Siendo la 'A' el 10, la 'B' el 11 y sucesivamente hasta la 'F' el 15. Ej: `int myValue = 0x1A3; //419 en decimal.`

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 implementarlas. 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. El uso de punteros podría tomarse como operación pero por el momento no vamos entrar en ello.
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.`

Truco:

Cuando se trabaja con constantes numéricas sin decimales, estas siempre se tratan como enteros en base 10. Si prestamos atención al ejemplo del tipo long *visto anteriormente:* long amount = -34254329L; *aparece una* L *al final de la constante numérica; es para forzar al compilador para que la trate como un* long *en lugar de como un* int.
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 el 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 tipos booleanos, sólo aceptan `true` y `false` como valores. No se deben confundir con las operaciones a nivel de bit. 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. Sólo 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 operan 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 sólo 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 bien 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. Se ofrece más información sobre operaciones a nivel de bit en los apéndices.
Las operaciones compuestas son aquellas que con una operación podemos hacer dos, por ejemplo sumar y asignar.
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 ver. `myVar *= 10;` es lo mismo que `myVar = myVar * 10;`
- /=: Asignación con división. Permite dividir y asignar a la ver. `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. En este apartado se verán los bloques de control de flujo que nos servirán para indicar al programa que realice una acción un número determinado de veces, controlar cuando ejecutar unas acciones u otras dependiendo de algo, etc.

if

El bloque `if` permite ejecutar una sección de código cuando se cumpla una condición dada. La estructura es:

```
if (condición){
   //código a ejecutar
}
```

La condición dada entre los paréntesis 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 solamente una línea, entonces las llaves se pueden omitir, o dicho de otro modo, en caso de no poner llaves, sólo se ejecutaría la primera instrucción tras el `if` (hasta encontrar un ";"). Personalmente siempre aconsejo utilizar las llaves aunque el bloque de condición sea una sola línea, ya que si el día de mañana se añaden nuevas líneas, al no tener las llaves ya puesta, se suele olvidar colocarlas para que acepte las nuevas líneas y no tendrá el comportamiento esperado. Si se decide trabajar sin llaves, también es preferible usar la notación:

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

que:

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

porque al estar en una sola línea queda claro que el bloque `if` es de una instrucción. Por ejemplo para ejecutar un código siempre que la variable `a` sea mayor que `b` sería:

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

Advertencia:

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 siembre como verdadero a no ser que se esté asignando el valor 0; la sentencia* if (i=5){...} *siempre se ejecutará por ser* i *distinto de 0.*

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 entra en uno de los dos bloques sólo y siempre entra en uno de ellos. El bloque `else` funciona del mismo modo que el `if`, ejecutando lo que encuentra entre llaves o sólo 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
}
```

Existe la posibilidad de trabajar con una sintaxis abreviada del bloque `if... else` especialmente pensada para casos en que los bloques sean de una instrucción quedando una estructura muy reducida y normalmente de una sola línea. Su sintaxis es:

```
condicion?instrucción_IF:instruccion_ELSE;
```

Por ejemplo para asignar a la variable `x` el valor mayor entre `a` y `b` sería:

```
x= (a>b)?a:b;
```

Incluso se pueden anidar, en este ejemplo se asigna a la variable `x` el valor mayor entre `a` y `b` y 0 si son iguales:

```
x= (a>b)?a:(b>a)?b:0;
```

Pero como podemos observar se puede complicar su lectura así que a lo largo del libro se utilizará esta forma en contadas ocasiones de modo que no pueda causar confusión.

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. Suponga que tenemos una variable con el día de la semana y queremos sacar por pantalla el nombre del día de la semana dependiendo del valor que tiene dicha variable.
Se podría usar un `if... else` con seis comprobaciones más un `else` "por defecto"; en lugar de esto se puede usar el `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. Su sintaxis sería:

```
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 comience una nueva etiqueta no quiere decir que acabe el código de la etiqueta anterior, sino que se debe poner un `break` o seguirá ejecutando código. Por ejemplo:

```
int a = 10;
switch (a){
   case 1:
      código1 //si a igual a 1
      break;
   case 2:
   case 3:
      código2 //si a igual a 2 o 3
   case 5:
      código3 //si a igual a 5
      break;
   default:
      //código4 para el resto de valores de a
}
```

En el ejemplo anterior tal y como está el código se ejecutaría el codigo4, correspondiente a la etiqueta `default`; si `a` tuviera el valor 2, ejecutaría el código2 y código3 ya que al no haber un break; seguiría ejecutando el código marcado por la etiqueta 5 hasta llegar al `break`. En este caso, la ejecución sería idéntica si el valor de `a` fuera 2 o 3.

goto

Esta instrucción permite saltar a una parte de código previamente marcada con una etiqueta. Su sintaxis 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.

for

Sirve para repetir la ejecución de un bloque de código n veces. Usualmente este tipo de bloque cuenta con un contador para controlar el número de veces que se ha repetido el bucle. Su sintaxis 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. El ciclo de ejecución quedaría del siguiente modo, se ejecuta la `inicialización`, como comienza un nuevo ciclo (el primero) del bucle, 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 o se encuentre un `break`.

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

En el uso habitual del `for`, la `inicialización`, la `condición` y la `instrucción` están informadas, pero no es obligatorio que lo estén, puede haber dos, una o incluso ninguna de ellas informadas, eso sí, los ';' de separación son obligatorios en todos los casos. Un caso extremo sería no tener ninguno de los campos, entonces tenemos que proveer algún método para que el bucle pueda terminar y no nos encontremos ante un bucle infinito.

```
int j = 0;
for (;;){
   if (j == 10){
      break;
   }
   //código a ejecutar 10 veces
   j++;
}
//código2
```

En el ejemplo anterior también nos hemos valido del comando `break;` que se había utilizado en el bloque de control `switch` y que servía para detener la ejecución del bloque en cuestión, en este caso sale del bloque del `for` y continuaría en `código2`.
El bloque `for` es muy versátil y permite cosas como que en la parte de instrucción se puedan incluso poner más de una sentencia (separándolas mediante comas) o inicializar varias variables a la vez. Por ejemplo para aumentar la variable `j` y disminuir la `i` en cada ciclo sería:

```
for (int j = 0,  i = 10; j < 10; j++, i--){
//código
}
```

Al ser muy fácil implementar un índice contador de ciclos, el bucle `for` es muy utilizado para recorrer arrays. Para abandonar el ciclo actual y comenzar un nuevo ciclo podemos utilizar la instrucción `continue`, que lo que hace es precisamente eso, cuando durante la ejecución del bucle el proceso se encuentra con el `continue;` se finaliza el ciclo en ese preciso momento y continúa con un nuevo ciclo tras haber realizado las instrucciones pertinentes y realizar la comprobación definidas en la cabecera de `for`. No hay que confundirlo con `break` que salía del bucle por completo, esta instrucción termina la iteración y continúa el bucle con la siguiente. Por ejemplo para ejecutar un código sólo los ciclos impares podemos hacer que se ejecute la instrucción `continue` en los ciclos pares:

```
for (int i = 0; i < 10; i++){
   if (i%2==0){
      continue;
   }
   // código a ejecutar sólo los ciclos impares
}
```

while

Se trata de otro tipo de bucle cuya ejecución se mantiene activa hasta que la condición dada en la cabecera sea falsa o se encuentre una instrucción `break` durante su ejecución.

Su sintaxis 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`. 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.

```
int i = 10;
while (i >0){
   // código
   i--;
}
```

do...while

Es otro tipo de bucle muy semejante al bucle `while` con la excepción de que en este tipo la condición se comprueba al final del bucle, cada vez que se acaba el bucle mira si tiene que empezar uno nuevo mientras que en el `while` se comprueba antes de iniciar cada bucle. Su sintaxis:

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

El hecho de que se compruebe la condición al final de cada bucle asegura que los bloque `do...while` **al menos se ejecutan siempre una vez**.

```
int i = 10
do{
   // se ejecutará una vez
} while (i < 5);
```

Funciones

Una función es un bloque de código que realiza una tarea particular. Trabajar con funciones es una buena manera de organizar el código y crear 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, sólo esto y específicamente esto y no debe mezclarse con el resto de código" y hacer el código reusable en diferentes partes del código.

Es como generar cajas negras que dados unos parámetros devuelven un resultado o realizan tareas. Para definir una función se utiliza la estructura:

```
tipo_dato nombre_funcion(tipo_parametro1 nombre_parametro1){
   // código de la función
}
```

Donde `tipo_dato` es el tipo de dato que devuelve la función (si la función no devuelve nada, el tipo de dato es `void`), `nombre_función` es el nombre de la función que hayamos decidido darle, `tipo_parámetro1` es el tipo del primer parámetro de la función y `nombre_parámetro1` es el nombre que tendrá el parámetro dentro de la función como variable.
Los nombres de función y parámetros pueden ser los que queramos siempre que no sean palabras reservadas por el lenguaje Arduino (no podemos llamar a una función `long`), que comiencen por una letra y que su nombre sea compuesto por letras, números o bien el carácter '_'. Los nombres siempre tienen en cuenta las mayúsculas y las minúsculas, por lo que se tiene que tener cuidado al escribirlos. Puede que la función no tenga parámetros en este caso se queda vacío entre los paréntesis.

```
void changeStatus(){
...
}
```

Para llamar a las funciones vale con poner el nombre de la función, rellenar los parámetros en caso de que los tenga y asignar el resultado a un valor si es que devuelve algo distinto a `void`. Por ejemplo definamos una función que devuelva el valor mayor de los dos parámetros dados de tipo `int` y en caso de ser iguales devolver el segundo parámetro.

```
int getGreater(int val1, int val2){
   int returnValue; //variable interna
   if (val1 > val2){
      returnValue = val1;
   }
   else{
      returnValue = val2;
   }
   return returnValue;
}
```

En la definición de la función se especifica que devolverá un tipo entero, que la función se llama `getGreater` y que tiene dos parámetros `val1` y `val2`. Dentro de la función se define una nueva variable llamada `returnValue` que será la que recoja que valor se debe retornar como mayor de los dos pasados como parámetros. Se realiza la asignación de `val1` o `val2` dependiendo de cual es mayor a la variable `returnValue` y se devuelve ésta. Se podría haber realizado sin utilizar la variable intermedia `returnValue` ya que así

ahorramos memoria pero para claridad de lectura del código se recomienda tener un sólo punto de retorno dentro de las funciones. En caso de ser crítica la gestión de memoria se podrían sustituir las asignaciones por retornos.

```
int getGreater(int val1, int val2){
   if (val1 > val2){
      return val1;
   }
   return val2; // solo se ejecuta si val2 => val1
}
```

Para llamar a esta función se debería utilizar algo como:

```
int maxVal = getGreater(123, 2);
```

Tras esta llamada la variable `maxVal` tendrá el valor 123 que es lo que devolvería la función.

Volviendo a la estructura general del programa que se había visto anteriormente, se tienen dos funciones preestablecidas como esqueleto del programa, la `setup()` y la `loop()`; del mismo modo existen otras muchas funciones ya dadas por Arduino y que se irán viendo a lo largo del libro según se vayan utilizando. Por el momento, nos vale con saber que la función `setup()` nos permite configurar la tarjeta y la función `loop()` será la encargada de ir llamando a nuestra lógica de programa, que habitualmente se encontrará dividida en funciones.

Cuando se llama a una función, el proceso de la función desde la que se ha llamado se detiene, salta a la función llamada, y al terminar la ejecución de esta nueva función retorna.

El orden de la definición de las funciones es indiferente, es decir no hace falta tenerlas definidas antes de utilizarlas como pasaba antiguamente en C, aunque en muchos de los ejemplos de Arduino primero se definen todas las funciones y luego se usan, realmente no hace falta, las podemos definir en el orden que más nos interese. Véase la figura 2.1.

Los programas en Arduino se ejecutan de manera lineal instrucción a instrucción. Una vez comenzada la función `loop()`, irá ejecutando sucesivamente las instrucciones que vaya encontrando en el programa y una vez termina una instrucción se ejecuta la siguiente (sean instrucciones aisladas o funciones completas) y cuando terminan todas se vuelve a ejecutar el bucle definido por la función `loop()` desde el principio, pero existe la posibilidad de realizar múltiples tareas a la vez. Este método de trabajo se conoce como *multithread* o multihilo pero esta desaconsejado por los propios creadores de Arduino por crear más problemas y trabajo del que resuelven; prácticamente todos los problemas que necesitan multitarea en Arduino pueden resolverse en monotarea simplemente modificando la lógica del programa.

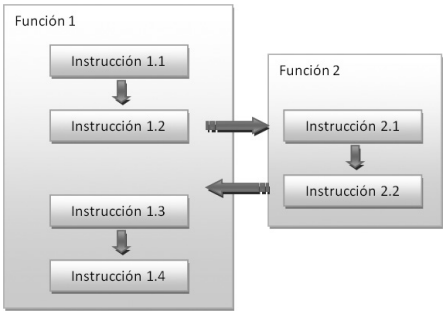


Figura 2.1. Esquema de ejecución de las funciones.

include y define

Tanto las instrucciones `#include` como `#define`, las encontramos al principio de los ficheros de código y modifican el comportamiento de ellos.
Mediante la instrucción `#include` hacemos disponibles al *sketch* librerías externas tanto propias como de terceros, haciendo posible realizar llamadas a sus funciones desde cualquier parte del código. Para incluir la librería con funciones de acceso al uso de tarjetas SD se utilizaría:

```
#include <SD.h>
```

Por otra parte la instrucción `#define` permite definir constantes globales. Aunque ya se ha visto otra manera de crear constantes esta es similar en resultado pero distinta en proceso. Al utilizar la instrucción `#define`, el compilador cuando compila el programa sustituye todas las ocurrencias de la constante definida por su valor, mientras que con el modificador `const` no lo hace y será durante la ejecución donde cada vez que se necesite el valor de la constante vaya a memoria a buscarlo; el uso de `#define` reduce el uso de memoria y de tiempos durante la ejecución del programa, no obstante es más seguro para evitar errores utilizar el modificador `const`. La sintaxis es:

```
#define constantName value
```

Por ejemplo para definir la constante días de la semana sería:

```
#define daysOfWeek 7
```

Nota:

Al utilizar tanto #define *como* #include *no se debe terminar la instrucción mediante ';'.*

Juntando las piezas

Con todo lo visto durante este capítulo ya podemos ser capaces de hacer pequeños programas, pero antes de lanzarnos a la aventura, vamos a analizar en profundidad el *sketch* de ejemplo que se usó en el capítulo anterior.
Si no lo hizo ya abra el entorno de desarrollo Arduino y cargue el sketch *Blink* mediante Archivo>Ejemplos>Basics>Blink. El resultado es que se abrirá una nueva ventana del entorno con el contenido:

```
/*
Blink
Turns on an LED on for one second, then off for one second, repeatedly.

This example code is in the public domain.
*/

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

// the setup routine runs once when you press reset:
void setup() {
   // initialize the digital pin as an output.
   pinMode(led, OUTPUT);
}

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

Veamos parte por parte lo que tenemos en este *sketch*. Lo primero que podemos encontrar es un comentario multilínea dando información sobre el cometido del programa y su licencia y otro comentario acerca de la utilización de la variable `led`. Mediante:

```
int led =13;
```

Se define la variable `led` para que tenga el valor 13. El led que viene incluido en la placa Arduino Uno está ligado al pin 13 y es por eso que le damos este valor a la variable y así más adelante poder referenciar el pin al que queremos enviar la señal para encender el led.

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

La función de `setup()` se encarga de configurar la tarjeta para trabajar con ella, en este caso se prepara el pin dado por la variable `led` (el pin 13) para trabajar en modo de salida. La función `pinMode()` es una función estándar de Arduino que permite configurar los pines de la tarjeta para trabajar en modo de entrada (`INPUT`) o de salida (`OUTPUT`). Como el programa lo que quiere es encender un led, configuramos el pin como salida (damos información al exterior, no la recibimos). Por defecto, los pines están configurados para trabajar en modo de entrada, por lo que no es necesario configurarlos nuevamente, pero para facilitar la comprensión de los programas se recomienda realizar dicha configuración explícitamente. Al configurar el pin como de salida, lo que se hace es ponerlo en baja impedancia. Los pines de Arduino pueden proveer o consumir hasta 40 mA de corriente cuando se conectan a otros circuitos o dispositivos lo que nos da una intensidad suficiente para alimentar un led.

Una vez se configuran los pines como salida/entrada se debe tener cuidado al realizar las conexiones con los cables de no producir cortocircuitos que puedan dañar la placa.

Unas líneas más adelante encontramos la función `loop()` que es donde se encuentra toda la lógica de funcionamiento del programa. En este caso como son muy pocas líneas de código, se han introducido en este bloque todas las instrucciones necesarias. Para encender el led se utiliza la instrucción:

```
digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
```

La función `digitalWrite()` permite indicar en qué estado se tiene que poner un pin; mediante dos parámetros, la función sabe qué número de pin se trata (en nuestro caso el 13 dado por la variable `led`) y el valor que debe tomar este pin, en este caso el valor `HIGH`. Al estar en modo `HIGH` significa que el pin estará con una tensión de 5 voltios (o 3.3 voltios dependiendo del modelo de placa), en modo `LOW` la tensión será de 0 voltios.

```
delay(1000);               // wait for a second
```

Mediante la función `delay()` se puede crear una pausa en el programa, éste se detiene y espera los milisegundos indicados como parámetro antes de continuar la ejecución de la siguiente instrucción.

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

Por último se vuelve a modificar la tensión del pin para ponerlo en modo `LOW`, a 0 voltios y se vuelve a esperar un segundo. Cuando se ejecuta de modo seguido el programa lo que se obtiene es un led que se enciende y apaga estando un segundo en cada estado.

Primer programa

Vamos a crear nuestros primeros programas y así comenzar a experimentar con lo aprendido en este capítulo. Para estos ejemplos no utilizaremos nada más que la placa Arduino, dejando para los siguientes capítulos el uso de más material electrónico.

Para ver el funcionamiento de los programas nos valdremos del monitor serie que se incorpora en el entorno de desarrollo, desde el cual podremos tanto obtener como suministrar información a la placa. En capítulos posteriores se entrará más en detalla de la comunicación serie y del monitor, pero veamos unas pinceladas del tema.

Para poder trabajar con el monitor serie, lo primero que se ha de hacer es configurar la conexión dentro del bloque correspondiente a la función `setup()`, esto se hace mediante `Serial.begin(velocidad);` donde la velocidad es la tasa de transferencia que se utilizará medida en bps (bits por segundo). Las tasas de transferencia también se pueden encontrar medidas en baudios ya que al estar hablando de comunicaciones serie en bits coinciden en número.

En cuanto al número, para comunicarse con el PC podemos utilizar 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, o bien 115200 bps (seguro que más de uno ahora recuerda su antiguo modem), pero la comunicación puede realizarse también con otros dispositivos por lo que técnicamente se puede poner cualquier valor con tal de que la tasa de transferencia se ajuste a la velocidad necesaria. Igualmente si se quiere desconectar la comunicación serie, la función a la que se debería llamar sería `Serial.end();`.

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

Una vez configurada la conexión ya puede ser utilizada dentro del bucle para transmitir datos al monitor. Para escribir en el monitor podríamos utilizar `Serial.print()` o `Serial.println()`.

La llamada a `Serial.print()` tiene la sintaxis:

```
Serial.print(valor,formato);
```

siendo el parámetro `formato` opcional. El valor puede ser cualquier tipo de dato usado en Arduino, mientras que el formato sirve para indicar la base en la que se quiere mostrar el valor si éste es de tipo entero o el número de decimales para los tipos de punto flotante. La función retorna el número de bytes escritos durante la transmisión, pero su utilización es opcional.

`Serial.println()` se utiliza de la misma mamera que `Serial.print()` y se diferencian en que `Serial.println()` añade también los caracteres de retorno de carro y nueva línea cada vez que se ejecuta. Por ejemplo:

```
int  year = 1992;
Serial.print("1992 en hexadecimal se escribe:");
Serial.println(year, HEX);
```

Vamos a usar parte del código anterior para ver si funciona; cree un nuevo sketch mediante el menú Archivo>Nuevo e introduzca el código:

```
void setup() {
   // se configura el puerto serie para trabajar a 9600 bps
   Serial.begin(9600);
}
void loop() {
   int  year = 1992;
   Serial.print("1992 en hexadecimal se escribe:");
   Serial.println(year, HEX);
   delay(500); // se detiene el programa medio segundo antes de un nuevo
               // ciclo.
}
```

Con la tarjeta conectada al PC y modelo y puerto configurados, pulsamos el botón para cargar el *sketch* en la tarjeta tal y como se hizo anteriormente. Si pulsa sobre el botón del monitor serie  o bien en el menú Herramientas> Monitor serial, se abrirá la ventana de monitorización donde en breves instantes veremos aparecer la frase "1992 en hexadecimal se escribe: 7C8";
Vamos a ser un poco más exigentes y vamos a crear un convertidor de decimal a otras bases donde tanto el número a convertir como la base a la que convertir sean introducidas por el monitor serie. Comience un nuevo *sketch*.
Lo primero a realizar en este nuevo *sketch* es crear dos variables globales, una para mantener el número a convertir y otra para la base:

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

En la función de configuración preparamos la conexión serie a 9600 bps:

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

En el bucle principal, debemos obtener un número, convertirlo, mostrarlo por pantalla y volver a iniciar un nuevo ciclo. Para la entrada de datos usaremos dos funciones distintas, una que se encargue de obtener el número a convertir y otra distinta para obtener la base y realizar la conversión. Antes de finalizar el bucle inicializamos las variables para comenzar de nuevo.

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

Dentro de la función `getNumber()` se leerá la entrada serie y se obtendrá el número a transformar. Según Arduino va recibiendo datos a través de su puerto serie, los va almacenando en un buffer de tamaño 64 bits; mediante la llamada a `Serial.available()` se puede obtener el número de bytes o caracteres disponibles para leer. La clase `Serial` ofrece varios métodos para la lectura de este buffer, lo que realmente guarda el buffer son bytes, que en el caso de usar la ventana del monitor serie (como es este caso) para enviar los datos, lo que guardará es el valor ASCII del carácter introducido; entre otros métodos de ayuda para la lectura tenemos `Serial.parseFloat()` para leer y convertir lo leído en un tipo `float` y `Serial.parseInt()`, que lee y convierte a `int`. En este caso utilizaremos `Serial.parseInt()` para la lectura del número. Debemos tener en cuenta que va a devolver un tipo `int` sin signo, es decir el valor a convertir debe ser menor de 65536. Para evitar que el usuario no introduzca un valor o intente convertir el 0, usaremos una estructura `do...while` que tenga como condición que el número a convertir sea 0, es decir, que mientras sea 0 se seguirá intentando leer del puerto serie.

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

La función `getBaseAndConvert()` debe mostrar un mensaje para que el usuario sepa qué tipo de valor debe introducir dependiendo de la base a la que quiera convertir, obtener la base y convertir el número anteriormente captado en la base seleccionada. Los mensajes informativos ya se ha visto de qué forma realizarlos mediante `Serial.print()` y `Serial.println()`. La captura del tipo de base la haremos leyendo directamente del buffer

el primer carácter que tengamos, para ello usaremos el método `Serial.read()`, que precisamente cumple esta función, devolver el primer byte del buffer. Durante las lecturas, además de leer los bytes, estos se retiran del buffer, con lo que a la vez que se leen se va vaciando el buffer. Con esto quiero decir que si hay más de un byte y se lee el primero el resto de bytes quedarán en el buffer a la espera de que alguien los lea. Existe una manera de leer sin extraer el byte del buffer que ya veremos más adelante. Pero volvamos al código. Una vez obtenida la base (que será una letra), debemos comprobar que es una base válida, que en este caso será 'b' para binario, 'o' para octal y 'h' para hexadecimal, y cualquiera otra debe generar un error e informarse al usuario para que sepa que no es válida la base introducida y que tiene que meter una válida. Esto lo solucionaremos con un bucle que se encargue de leer del puerto serie mientras la base no sea válida y la validez de la base se hará mediante una estructura `switch` con las entradas correspondientes a las bases válidas y una entrada `default` para el resto. Si las entradas son válidas, entonces pondrá a `true` una variable booleana que será la que controle que puede salir del bucle.
Si la base leída es correcta, dentro de cada caso se mostrará por pantalla utilizando un `Serial.print()` con formato.

```
/**********************************
*  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 numero ");
   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);
}
```

Para probarlo, se carga el *sketch* en la tarjeta y se lanza el monitor serie; al cabo de unos instantes aparecerá un mensaje pidiendo que se introduzca un número; en la parte superior de la pantalla se introduce el numero a convertir y para enviarlo al puerto serie se debe pulsar en el botón **Enviar**.
Más adelante se nos 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 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, dando errores para el resto.

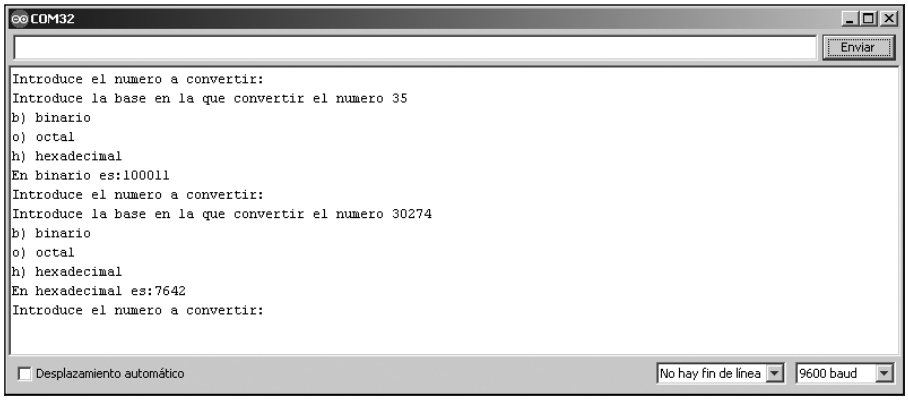


Figura 2.2. Monitor serie con la salida del sketch.

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 numero 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 numero 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 numero ");
   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);
}
```

3 Introducción de datos analógicos y digitales

En este capítulo aprenderá a:

- Diferenciar entre datos analógicos y digitales.
- Configurar los pines de entada de Arduino.
- Leer y detectar estados de las entradas digitales.
- Obtener lecturas de voltajes mayores de 5V.

Las placas Arduino ofrecen un número de pines de entrada por los cuales se puede informar de la situación del mundo exterior con tal de que el microcontrolador pueda tenerlo en cuenta para actuar de diferentes maneras. Supongamos que queremos hacer un timbre; el microcontrolador deberá hacer que suene la campana cuando el usuario pulse un botón y deje de sonar cuando lo suelte, entonces deberemos ser capaces de "leer" el dedo del usuario pulsando un botón para saber que quiere llamar y reaccionar a los cambios de presión de ese botón; a lo mejor queremos que para que no nos quemen el timbre éste debe de dejar de sonar cuando lleve 30 segundos apretado el botón, para lo cual el microprocesador debe de obviar la entrada pasado ese tiempo aunque el dedo siga pulsando el botón de timbre. Para poder actuar así y que la placa tenga constancia de los cambios en el mundo real, debemos ser capaces de llevarlo al campo eléctrico, haciendo que alguna de las entradas de Arduino se vea afectada por estos cambios.
En este capítulo se comenzarán a realizar ejercicios con montajes físicos; debemos poner atención en realizar de modo correcto las conexiones y las polarizaciones de los elementos para que la tarjeta no sufra daños.

Entrada digital y analógica

Cuando se habla de entradas digitales, debemos pensar en que pueden tener sólo dos valores, encendido y apagado, 1 y 0, `true` y `false`, `on` y `off`, `HIGH` y `LOW` y otras muchas maneras de nombrarlos. Internamente será como si hubiera un interruptor que marcara si está encendido o apagado. Realmente no recibimos por parte del mundo exterior un `LOW`, sino que se reciben tensiones eléctricas que deben ser interpretadas como encendido o apagado. Como convención general se toma que la representación de apagado son 0 voltios, mientras que la de encendido es 5 voltios, aunque existen placas que funcionan a 3.3 voltios como valor de encendido incluso algunas de lógica negativa donde los 5V son el 0 u otras combinaciones, ya que no deja de ser una convención.
Actualmente los microsistemas tienden a trabajar con voltajes menores porque esto significa una menor disipación de potencia y por lo tanto generan menos calor y existe menor consumo, pero en el caso de Arduino Uno trabajaremos a 5 voltios. Realmente no quiere decir que el valor `LOW` tengan que ser 0V exactamente y el `HIGH` 5V, ya que es prácticamente imposible tener esos valores exactos por caídas de tensión en los materiales y por ruidos e interferencias externas que hacen que las tensiones fluctúen. En lugar de eso se toma como referencia un valor y cuando el voltaje es mayor de ese valor se

tomará como encendido y cuando sea menor se tomará como apagado, y se trabajará lo más lejos posible de ese valor para que aunque haya interferencias no haya cambios de valor. Supongamos que el valor de referencia es 2.5V. Si el circuito alimenta a 2V será tomado como valor 0 en digital, pero si existe un pequeño acoplamiento de señal de 0.6 voltios, entonces la señal pasaría a ser de 2.6V por lo que se tomaría como un valor digital 1 y produciría error de transmisión del dato. Si se trabaja a 0V como valor 0, podemos tener interferencias de hasta 2.5 voltios.

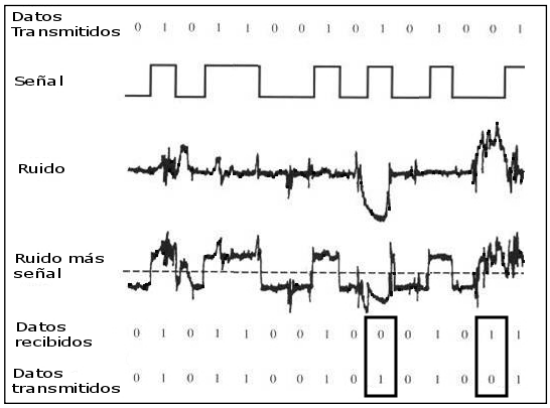


Figura 3.1. Error en el valor por distorsión de voltajes.

El valor de referencia en Arduino son los 3 voltios, por lo que cuando se sobrepase ese valor de entrada se tomará como que el pin está en `HIGH` y si el voltaje de entrada está por debajo se tomará como `LOW`. Para leer las entradas digitales, usaremos la función `digitalRead(pin);` donde `pin` es el número de pin digital sobre el que leer y retornará los valores `HIGH` o `LOW` en función del voltaje recibido.

Si en cambio hablamos de entradas analógicas, los valores pueden ser múltiples. En analogía al interruptor de dos posiciones que hace que la bombilla esté encendida o apagada, existen otros interruptores que hacen que la bombilla luzca más o menos, y tienen múltiples posiciones. Cuando Arduino recibe una entrada analógica lo hará con un voltaje entre 0 y 5 voltios, que será transformado en los valores 0 y 1023 respectivamente para trabajar dentro del microcontrolador. 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, así si en el pin de entrada existen 2.3 voltios, el valor recibido sería 2.3V*1023/5V = 471 y a partir de este valor es el programa quien tiene que interpretarlo

para saber si son 20°C o 38°C en caso de que estemos usando un termómetro para alimentar esa entrada. Muchos de los sensores que veremos a lo largo de los ejemplos proporcionan sus lecturas a la placa Arduino a través de las entradas analógicas.

Para 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 nada en la entrada; es decir en ausencia de 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 la resistencia se encuentra en circuito abierto en la rama de 5 voltios se denomina *pull-up* y si se encuentra en la rama de 0V de *pull-down*.

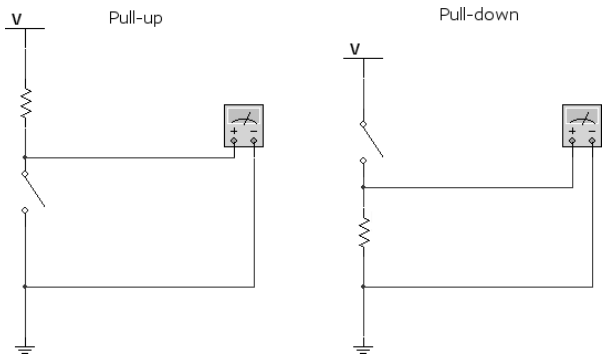


Figura 3.2. Resistencias de pull-up y pull-down.

En caso de no querer utilizar resistencias de *pull-up* externas, Arduino incorpora unas internas de 20k y para activarlas se utilizaría el código:

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

Disposición de pines de entrada

La disposición de los pines depende mucho del modelo de tarjeta que se vaya a utilizar por lo que es conveniente revisar la documentación de cada una de ellas. Para el caso de la Arduino Uno se disponen de 14 pines digitales de entrada/salida numerados de 0 a 13, también conocidos como pines de salida entrada de propósito general o GPIO (*general purpose input/output*). Los pines 0 y 1 están marcados como RX (recepción) y TX (transmisión) y

son utilizados para las transmisiones serie por lo que se deben intentar no usar (siempre que sea posible). La mayoría de los modelos llevan asociado en el pin 13 un led integrado (una bombillita), lo que hace que se comporte distinto al resto cuando trabaja en modo entrada. Si se utiliza la resistencia de *pull-up* interna, esta entrada obtiene una lectura menor de 2V en lugar de los 5V esperados debido a la caída de tensión en el led, esto quiere decir que `digitalRead(13);` siempre dará `LOW`, por lo que el montaje del *pull-up* se debe hacer con resistencias externas.
También se cuenta con 6 entradas analógicas con una resolución de 10 bits lo que da un resultado analógico de 1024 (2^{10}) valores posibles (de 0 a 1023). Los pines analógicos disponen también de resistencias de *pull-up* y pueden ser configurados a su vez como pines digitales extendiendo los 14 pines GPIO.

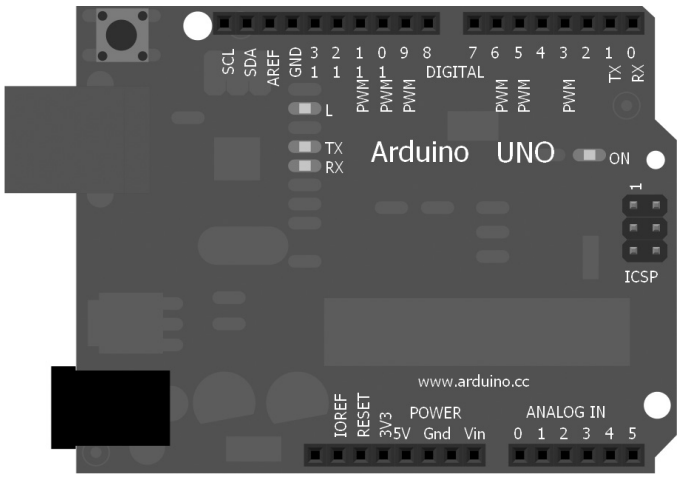


Figura 3.3. Esquema de la placa Arduino Uno.

Existen otros pines que tienen una vírgula (~) como marca al lado del número del pin; estos pines soportan el uso de modulación por amplitud de pulso o PWM (*pulse width modulation*), que sirve para simular salida analógica desde pines digitales. Más adelante entraremos en más detalle sobre ello.
Los pines de los diferentes modelos de placas suelen estar en la misma disposición que los mostrados en la figura anterior, aunque es posible que cambien la posición sobre todo los de las entradas analógicas. Manteniendo los pines en la misma posición que la placa Arduino Uno, se asegura que las *shield* son válidas para más modelos de tarjetas, aunque antes de comprar una *shield*, hay que asegurarse que es válida para el modelo de tarjeta con el que se va a trabajar.

Entradas digitales con resistencias pull-up externas

Para poder realizar las lecturas de las entradas digitales, lo primero que debe hacerse 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. Sencillo ¿no?, pues manos a la obra. La primera lectura la realizaremos mediante un montaje con una resistencia de *pull-up* externa. Como componentes adicionales usaremos la protoboard que se utilizará en todos los circuitos del libro, una resistencia de 10kΩ (ver apéndices), un pulsador y cables que también serán necesarios en la mayoría de circuitos que se vena a lo largo del libro. En este ejemplo se quiere mostrar en la consola serie el estado del pulsador que estará conectado al pin de entrada número 4 (aunque podría haberse seleccionado otro cualquiera).

Un pequeño inciso antes de seguir con el ejemplo para comentar el funcionamiento de los pulsadores.

Los pulsadores son unos pequeños botones con 2 o 4 patillas que nos permiten trabajar con ellos a modo de interruptor, haciendo que un circuito se cierre o se abra cuando éstos se pulsan. Los más comunes son aquellos que cuando no se pulsan hacen que el circuito permanezca abierto, cerrándolo al ser pulsados; pero también existen que funcionan de modo contrario, abriendo el circuito al pulsarlos y manteniéndose cerrados si no hay pulsación. En el caso de contar con 4 patillas (más cómodos para trabajar con la protoboard), éstas se encuentran unidas por pares aunque esté abierto el circuito, quedando todas ellas unidas cuando el circuito se cierra. El esquema se puede ver en la figura 3.4 donde se han marcado las patillas que se encuentran unidas.

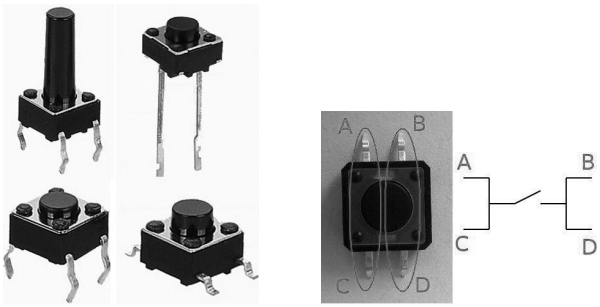


Figura 3.4. Pulsador electrónico.

Realizaremos el montaje como el de la figura 3.5. Como consejo es mejor utilizar un código de colores en los cables utilizados (siempre que sea posible por disponibilidad de ellos), por ejemplo rojo para alimentación, negro para tierra (0V)...

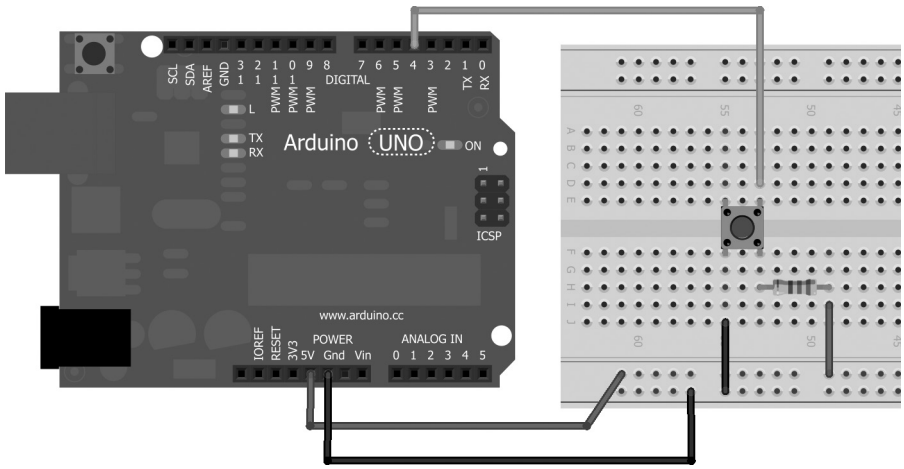


Figura 3.5. Circuito con pull-up.

El código correspondiente sería:

```
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 == LOW) {
      Serial.println("Pulsado");  // se encuentra pulsado
   }
   else{
      Serial.println("Libre!!!!");  // no está pulsado
   }
}
```

Lo que hacemos en el código es configurar la placa para que trabaje con el pin 4 como entrada y preparar la comunicación serie. Una vez está configurada, en el bucle principal se va leyendo la entrada 4 y dependiendo del valor obtenido se muestra una salida u otra por pantalla. La comprobación del valor leído se realiza mediante:

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

Se compara lo leído con `LOW` para determinar si está pulsado, ya que al ser un montaje mediante resistencia de *pull-up*, si no está pulsado, es decir a "interruptor abierto" tendrá valor de entrada `HIGH` fijado precisamente por esta configuración.
Si cargamos el programa en la placa Arduino y después seleccionamos el monitor serial, veremos cómo van saliendo mensajes del estado de pulsación del botón, en este caso **"Pulsado"** cuando se encuentre pulsado y **"Libre!!!"** cuando no lo esté.
Si por el contrario realizamos el montaje con resistencia de *pull-down*, lo que estamos haciendo es que cuando no haya señal en la entrada, es decir cuando el interruptor se encuentre abierto, se fuerce la entrada a 0V. Las conexiones para la configuración en *pull-down* es la mostrada en la figura 3.6. Si ejecutamos el *sketch* anterior con esta nueva configuración, se obtendrá en la salida justo lo contrario de lo esperado, ya que cuando no se pulse el botón, el valor recibido será `LOW`. Para que funcione de modo correcto se debe cambiar la línea de código:

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

por:

```
if (val == HIGH) {
```

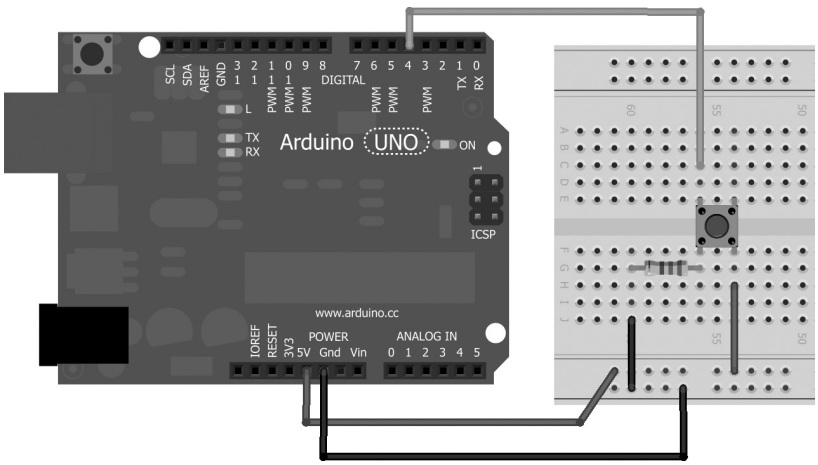


Figura 3.6. Circuito con pull-down.

Para hacer un poco más llamativo este primer ejemplo, en lugar de utilizar el monitor serie para mostrar el estado del botón, usaremos el led integrado, encendiéndolo cuando el botón se encuentre apretado. El montaje eléctrico será el mismo, mientras que el programa lo tenemos que variar para que

cuando sepa el valor del pulsador lo muestre en el led. Para poder escribir un valor en un pin se utiliza la función `digitalWrite(pin,valor);` donde pin es el número de pin al que se le quiere dar la salida y `valor` podrá ser `HIGH` o `LOW` dependiendo de si se quiere poner a 5V o a 0V.

```
const byte ledPin = 13; // pin del led
const byte inputPin = 4; // pin para el botón
void setup() {
   pinMode(ledPin, OUTPUT); // pin del led como salida
   pinMode(inputPin, INPUT); // se declara como pin de entrada
}

void loop(){
   int val = digitalRead(inputPin); // lectura del valor
   if (val == HIGH){
      digitalWrite(ledPin, HIGH); // se encuentra pulsado
   }
   else{
      digitalWrite(ledPin, LOW); // no está pulsado
   }
}
```

Dado que el valor de la entrada leída como el que se le asignado a la salida para encender o apagar el led es el mismo, podemos aprovechar la llamada a `digitalRead()` para hacer de entrada de la llamada `digitalWrite()`. La función `loop()` con esta nueva forma de trabajar funcionaría igual y quedaría mucho más compacta.

```
void loop(){
digitalWrite(ledPin, digitalRead(inputPin));    // asignar el valor leído
                                                // al led
}
```

Entradas digitales con resistencias pull-up internas

Para facilitar poder asignar el valor por defecto de las entradas digitales, Arduino pone a disposición de los usuarios unas resistencias internas que pueden ser activadas y ser utilizadas para configurar las entradas con resistencia de *pull-up*, es decir a circuito abierto su tensión es de 5V. Realizaremos el ejercicio anterior nuevamente, seguiremos encendiendo el led 13 al pulsar el botón, pero esta vez ahorrando componentes. Dado que la resistencia interna es de *pull-up* y tendemos 5 voltios en la salida, lo que se hará para detectar la pulsación es que el botón pulsado lleve a la entrada 4 la señal de tierra. El circuito quedaría como muestra la figura 3.7.

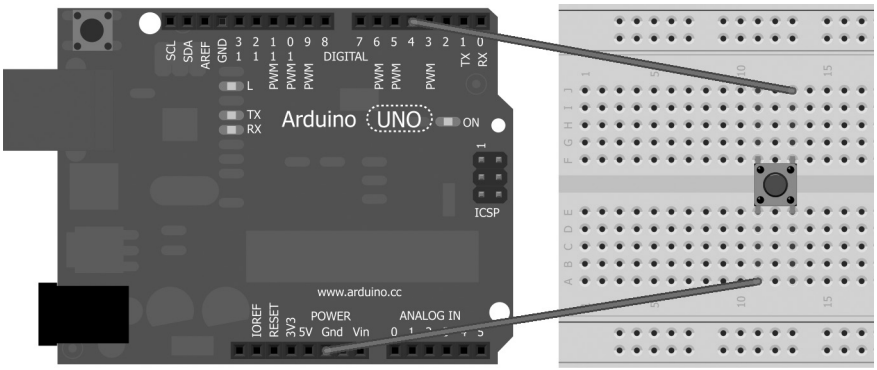


Figura 3.7. Circuito con pull-up interno.

El código podemos hacer una mezcla de lo ya visto anteriormente con la configuración relativa al *pull-up* interno para obtener:

```
const byte ledPin = 13; // pin del led
const byte inputPin = 4; // pin para el botón
void setup() {
   pinMode(ledPin, OUTPUT); // pin del led como salida
   pinMode(inputPin, INPUT); // se declara como pin de entrada
   digitalWrite(inputPin, HIGH); // se activa la resistencia de pull-up
}
void loop(){
   digitalWrite(ledPin, !digitalRead(inputPin));   // asignar al led el valor
                                                   // contrario al leído
}
```

También en el código hemos hecho cierto ahorro. Como queremos encender el led cuando esté apretado y en esa situación en la entrada hay 0V (que es un `false` o `LOW`) y necesitamos 5V (que es un `true` o `HIGH`), lo que hacemos es negar el valor devuelto por la lectura a la entrada digital. El operador *"!"* se encarga de esto en `!digitalRead(inputPin)`, así si lee `LOW`, en la salida escribirá `HIGH`.

Como este código ha quedado un poco corto vamos a complicarlo añadiéndole algo de funcionalidad. Puesto que es muy pesado estar manteniendo el botón para que se quede encendido el led, lo que haremos es que en cada pulsación encenderá o apagará el led. Nada más comenzar el programa el led aparecerá apagado, al pulsar (y sin necesidad de mantener pulsado) el led se encenderá y permanecerá encendido hasta que se vuelva a pulsar. Para poder implementar una solución a estas necesidades, necesitaremos una variable que "se acuerde" del estado en el que se encuentra el led y tener cuidado que si el usuario deja apretado el botón, el led no comience a encenderse y apagarse a lo loco.

```
const byte ledPin = 13; // pin del led
const byte inputPin = 4; // pin para el botón
boolean checkButton = true;

void setup() {
   pinMode(ledPin, OUTPUT); // pin del led como salida
   pinMode(inputPin, INPUT); // se declara como pin de entrada
   digitalWrite(inputPin, HIGH); // se activa la resistencia de pull-up
}
void loop(){
   if (!digitalRead(inputPin)){ // ver si esta pulsado
      if (checkButton){ // se debe atender la pulsación?
         checkButton = false;    // dejar de atender pulsaciones si se
                                 // atiende una
         switchLed(); // cambiar el led
      }
   }
   else{
      checkButton = true; // se ha soltado el botón, volver a atenderlo
   }
   delay(1);//pausa entre ciclos
}
/** cambia de estado el led **/
void switchLed(){
   static int ledStatus = LOW; // mantiene el estado del led
   ledStatus = !ledStatus; // cada pulsación lo cambia de estado
   digitalWrite(ledPin,ledStatus );
}
```

Vamos a poner una condición más y es que en lugar de que realice el cambio de encendido a apagado, como va tan cara la luz, para encender el led se debe mantener apretado un par de segundos mientras que para apagar valdrá con una pulsación corta. Es una manera de asegurarnos que no se enciende el led porque pulsemos sin querer.

Definiremos la variable `ledStatus` de manera global en lugar de estática dentro de la función para facilitar el código; la parte correspondiente a la función `setup()` quedaría idéntica.

En el `loop()` leeremos la entrada y miraremos si está pulsado, en caso de que lo esté hay que ver si se debe atender la pulsación o no (por ejemplo porque lleve más de 2s apretando) y en caso de que se deba atender puede ser que nos encontremos en dos situaciones:

- El led estaba encendido y acaban de pulsar el botón, por lo que hay que apagarlo sin esperar a los dos segundos y luego dejar de atender al botón hasta que se libere.
- El led estaba apagado y acaban de pulsar el botón, por lo que hay que encenderlo siempre y cuando se haya estado apretando más de dos segundos y luego dejar de atender al botón hasta que se libere.

```
int buttonStatus = !digitalRead(inputPin);
if (buttonStatus){ // ver si esta pulsado
   if (checkButton){ // se debe atender la pulsación?
      /* Comprobar si ya está encendido o lleva más de 2 segundos
         pulsado, si es así cambiar el estado del led*/
      if (ledStatus || checkTime(buttonStatus)){
         checkButton = false;    // dejar de atender pulsaciones si se
                                 // atiende una
         switchLed(); // cambiar el led
      }
   }
}
```

Mediante la comprobación `if (ledStatus || checkTime(buttonStatus))` comprobamos si se debe cambiar el estado del led; si está encendido y el usuario pulsa el botón, se debe apagar, con lo que es condición suficiente y en caso de estar apagado entonces debemos comprobar si lleva más de dos segundos apretado.

```
void switchLed(){
   ledStatus = !ledStatus; // cada pulsación lo cambia de estado
   digitalWrite(ledPin,ledStatus );
}
```

El cambio de estado del led, permanecería muy parecida a como estaba, salvo el hecho que ahora la variable `ledStatus` la hemos convertido en global. Para guardar el tiempo no podemos utilizar la función `delay()` ya que ésta bloquea el proceso completo, es decir bloquearía completamente la ejecución durante esos dos segundos y si se dejara de pulsar el microprocesador no se enteraría.

Esto sería sencillo de realizar en otros programas mediante *threads*, pero tal y como dijimos anteriormente, en Arduino no se aconseja su uso. Lo que se debe hacer es guardar el tiempo en el que se ha pulsado y en sucesivas llamadas comprobar si el tiempo actual menos el tiempo en el que se comenzó la pulsación es mayor a los 2000 milisegundos. La obtención del tiempo actual la haremos mediante la función `millis()`, que devuelve los milisegundos transcurridos desde el comienzo de la ejecución del programa; existe otra función semejante para cuando se necesita mayor granularidad y es `micros()`, que como se puede deducir, devuelve el número de microsegundos que lleva funcionando el programa. Ambas funciones están implementadas con un contador que llega un momento que se desborda y comienza de nuevo desde 0; en caso del `millis()` sucede cada 50 días y encaso de `micros()` cada 70 minutos, ambas aproximadamente.

También hay que diferenciar si cuando se llama a la función que controla el tiempo es que acaba de comenzar la pulsación o lleva varios ciclos de ejecución, esto lo haremos comparando la variable `butonStatus` que contiene el

estado actual del botón, con la variable `state` que guarda el último estado conocido, si son distintas indica que se acaba de comenzar una nueva pulsación o liberación del botón.

```
boolean checkTime(int buttonStatus){
   static unsigned long startTime = 0; // mantiene el tiempo pulsado
   static boolean state; // estado del led para saber si ha cambiado o no

/**
* si el estado del botón actual y el guardado no coinciden es que se
* ha dejado de apretar o se ha comenzado a apretar
**/
   if(buttonStatus != state) {
      state = buttonStatus; // se guarda el estado
      startTime = millis(); // se guarda el tiempo actual
   }
   if( state == HIGH){
      return (millis() - startTime)>2000; // lleva pulsado más de 2 s?
   }
   else{
      return false; // no está pulsado, no nos interesa;
   }
}
```

El código completo quedaría:

```
const byte ledPin = 13; // pin del led
const byte inputPin = 4; // pin para el botón
boolean checkButton = true; // si se debe atender el botón
boolean ledStatus = LOW; // mantiene el estado del led

void setup() {
   pinMode(ledPin, OUTPUT); // pin del led como salida
   pinMode(inputPin, INPUT); // se declara como pin de entrada
   digitalWrite(inputPin, HIGH); // se activa la resistencia de pull-up
}
void loop(){
   int buttonStatus = !digitalRead(inputPin);
   if (buttonStatus){ // ver si esta pulsado
      if (checkButton){ // se debe atender la pulsación?
      /* Comprobar si ya está encendido o lleva más de 2 segundos
      pulsado, si es así cambiar el estado del led*/
         if (ledStatus || checkTime(buttonStatus)){
            checkButton = false; // dejar de atender pulsaciones si se
                                 // atiende una
            switchLed(); // cambiar el led
         }
      }
   }
   else{
      checkButton = true; // se ha soltado el botón, volver a atenderlo
      checkTime(buttonStatus); // pone a 0 el tiempo presión
   }
   delay(1); // pausa entre ciclos
}
```

```
/** Cambia de estado el led **/
void switchLed(){
   ledStatus = !ledStatus; // cada pulsación lo cambia de estado
   digitalWrite(ledPin,ledStatus );
}

/**
* Guarda el tiempo que lleva el botón pulsado
* Devuelve true si lleva más de 2 seg pulsado, false en caso contrario
**/
boolean checkTime(int buttonStatus){
   static unsigned long startTime = 0; // mantiene el tiempo pulsado
   static boolean state; // estado del led para saber si ha cambiado o no

/**
* si el estado del botón actual y el guardado no coinciden es que se
* ha dejado de apretar o se ha comenzado a apretar
**/
   if(buttonStatus != state) {
      state = buttonStatus; // se guarda el estado
      startTime = millis(); // se guarda el tiempo actual
   }
   if( state == HIGH){
      return (millis() - startTime)>2000; // lleva pulsado más de 2 s?
   }
   else{
      return false; // no está pulsado, no nos interesa;
   }
}
```

Entradas analógicas

Hasta ahora hemos estado trabajando con unas entradas de las que podíamos recibir el valor `LOW` o `HIGH`, correspondiendo a los valores lógicos 0 y 1 respectivamente. Eléctricamente hablando, lo que se tiene a la entrada es una tensión de entre 0 y 5 voltios, que nosotros internamente hemos tomado como 0 o 1 dependiendo de si pasaba o no de un valor, pero por ejemplo nos daba igual si la entrada era de 0.2V o de 0.28V, para nosotros era un `LOW`. En las entradas analógicas sí hay que tener en cuenta esa pequeña diferencia.

Mediante las entradas analógicas vamos a poder leer un rango de valores que más adelante tendremos que interpretar para transformar el valor leído en valor útil. Por ejemplo si medimos distancias, el dato leído en la entrada tendrá que ser interpretado para obtener un valor en unidades de medida. Realmente en la entrada seguimos obteniendo una tensión que variará desde 0 hasta 5V y que en el caso de Arduino será transformada en un valor con una precisión de 10 bits para poder trabajar internamente de manera más cómoda; los valores posibles de voltaje hasta los 5V se distribuirán uniformemente

entre 0 y 1023. Por ejemplo podemos tener un sensor lineal de temperatura que mida desde los 10 a los 30 grados, si hace 10 grados de temperatura tendríamos 0 voltios a la entrada y si hiciera 30 grados tendríamos los 5 voltios.

Ahora bien... ¿qué temperatura sería si internamente la entrada marca 432? Lo veremos en el capítulo referente a los sensores, donde se realizará un termómetro, pero por ahora nos conformamos con entender que se deben hacer ciertas operaciones para poder saber el valor real que representa el voltaje obtenido.

Para jugar un poco con las entradas digitales usaremos un potenciómetro. El potenciómetro es una resistencia que el lugar de tener dos patas como las resistencias normales, tiene tres. Las dos patas de los extremos marcan la resistencia nominal, es decir el total de la resistencia del potenciómetro, mientras que la tercera pata es deslizante por la resistencia y sirve para modificar el valor de la resistencia entre esta pata y las patas que marcan el valor nominal, de manera que el valor de la resistencia entre las patas A y B aumentará a la vez que disminuye el de la resistencia entre las patas B y C, siendo la pata B la situada en el medio. Existen múltiples tipos de potenciómetros, de distintos tamaños, formas y usos; desde pequeños como los trimmer, lineales o *faders*, de tornillo... de manera que podemos encontrar el que más se ajuste al conjunto del circuito. Muchas veces se encuentran con uno de los terminales extremos a masa y el otro a tensión, de modo que la pata intermedia actúa como divisor de tensión entre los dos terminales extremos. Existe otro elemento muy parecido al potenciómetro y que se suele usar en montajes de potencia, son los reóstatos; básicamente son el mismo componente aunque el reóstato tan sólo tiene dos patas accesibles, una de las fijas y la móvil, quedando la tercera inaccesible y sin conectar a masa, de modo que no trabaja como divisor de tensión. Podemos trabajar con un potenciómetro como si fuera un reóstato simplemente no conectando una de las patas fijas.

Figura 3.8. Diferentes tipos de potenciómetros.

Alguno de los sensores que veremos a lo largo del libro llevan incorporado algún potenciómetro que tiene como utilidad calibrar el sensor.

Debido a que el potenciómetro trabaja dividiendo la resistencia nominal en dos, lo utilizaremos para realizar un circuito con la placa Arduino de modo que funcione como un divisor de tensión, leyendo la tensión de la pata central como entrada de datos, así, al variar el potenciómetro, la caída de tensión en la pata central variará.

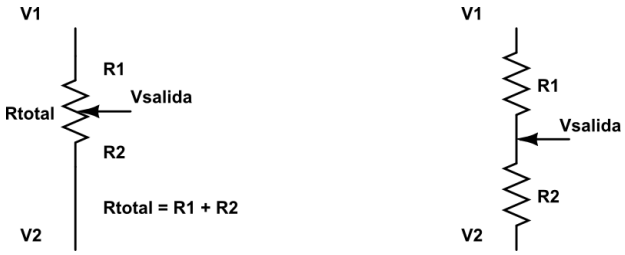


Figura 3.9. Funcionamiento del potenciómetro.

Vamos a ver ahora rápidamente de qué forma funciona el divisor de tensión. El voltaje obtenido depende de la intensidad y de la resistencia según la ley de Ohm:

$V = I * R$

Teniendo en cuenta la figura 3.9 se obtendría:

$V_1 - V_2 = R_{total} * I$

Si queremos calcular la tensión V_{salida} en lugar de V_1 -V_2, tenemos que tener en cuenta que la intensidad será la misma pero la tensión en ese punto correspondería solamente a la caída de tensión producida por la resistencia R_2 luego la fórmula para obtener la tensión suponiendo que V_2 es 0V sería la de la figura 3.10.

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

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

Figura 3.10. Fórmula para el divisor de tensión.

En el circuito conectaremos una de las patas fijas del potenciómetro al pin de alimentación de 5V de la placa Arduino, mientras que la otra pata fija la conectaremos a uno de los pines de tierra marcados con *GND* en la placa; da igual al que se conecte ya que todos ellos están unidos entre sí. La pata central del potenciómetro la conectaremos con la entrada analógica 0, marcada en

la placa como A0. Como ya sabemos cómo encender el led integrado de la placa, lo que haremos será crear un programa que encienda y apague el led con una frecuencia dependiente del valor leído en la entrada y sabiendo que la lectura que obtendremos en el programa Arduino será entre 0 y 1023, utilizaremos este valor como el tiempo que debe estar encendido y apagado el led.

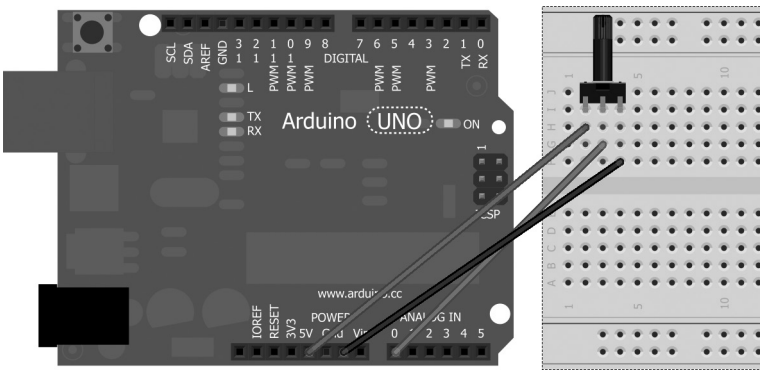


Figura 3.11. Circuito con potenciómetro en entrada analógica.

El programa sería:

```
const int inPin = A0; // pin de entrada
const int ledPin = 13; // pin del led
int val = 0;

void setup() {
   pinMode(ledPin, OUTPUT);
}

void loop() {
   val = analogRead(inPin); // se lee el valor del pin de entrada
   digitalWrite(ledPin, HIGH); // enciende el led
   delay(val); // espera el valor leido en la entrada en milisegundos
   digitalWrite(ledPin, LOW); // apaga el led
   delay(val);  // espera el valor leído en la entrada en milisegundos
}
```

El pin de entrada analógica puede ser referido como pin 0 o como A0. Puede usarse el 0 porque al realizar la llamada a la función `analogRead()` el sistema ya sabe que debe dirigirse a la entrada analógica, pero puede crear confusión cuando se utiliza a la vez que entradas digitales o cuando se usa la entrada analógica como puerto digital, por eso recomendamos el uso de la notación A0, es decir siempre que nos refiramos a una entrada analógica (aunque se use en modo digital) usaremos la A delante del número de entrada correspondiente. El programa es muy sencillo y con instrucciones

ya conocidas, pero cumple lo establecido, ahora para probarlo simplemente hay que variar el potenciómetro y observar como el led varía su velocidad de parpadeo.
Para conocer el valor que estamos obteniendo a la entrada, podemos crear un programa que muestre en el monitor serie cada una de las lecturas que realice. El siguiente *sketch* serviría:

```
const int inPin = A0; // pin de entrada
void setup() {
   Serial.begin(9600);
}

void loop() {
   int sensorValue = analogRead(inPin); // leemos el valor
   Serial.println(sensorValue); // lo mostramos en el monitor
   delay(1);        // pausa para estabilidad entre ciclos
}
```

Si cargamos el programa en la placa, veremos que al ejecutarlo y variar el potenciómetro, obtendremos en el monitor valores de 0 a 1023, como era de esperar. Pero en la vida real las escalas no son de 0 a 1023, sino que son muy variadas. Podemos tener un selector de volumen que vaya del 0 al 15 o a lo mejor puede que el valor mínimo del volumen sea 3 y que la escala sea de 3 a 15 (caprichos del fabricante)... ¿cómo se traducen estas escalas con el rango de valores obtenido? Lo más sencillo es usar la función `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. El valor devuelto es de tipo entero y siempre se trunca la parte decimal, es decir no existe redondeo de ningún tipo, en caso de necesitar redondeos habría que realizar los cálculos a mano.
Conociendo esto, podríamos sacar por pantalla el tanto por ciento de desplazamiento del potenciómetro modificando el código anterior añadiendo esta función:

```
void loop() {
   int sensorValue = analogRead(inPin); // leemos el valor
   sensorValue = map(sensorValue, 0, 1023, 0, 100); // transformación a %
   Serial.println(sensorValue); // lo mostramos en el monitor
   Serial.print("%");
   delay(1);        // pausa para estabilidad entre ciclos
}
```

La función `map()`, no comprueba que el valor a transformar esté dentro de los rangos establecidos por la escala proporcionada, sino que continúa la extrapolación para calcular el valor que le correspondería, por ejemplo si en el código anterior obtuviéramos en la entrada una lectura con valor 1550 (que no es posible), esto nos daría en el monitor como que el potenciómetro estaba al 151%. Está claro que en el ejemplo que se ha utilizado no podemos obtener valores mayores (ni menores) de la escala informada en `map()`, pero en otras situaciones sí es posible que se den valores fuera de escala, por ejemplo recuerdo que en el colegio daban medio punto por hacer el examen sin tachones y sin faltas de ortografía, pero si sacabas un 10 y lo tenias bien presentado no quería decir que tuvieras un 10.5... era un 10. Si por cuestiones de lógica de funcionamiento necesitamos restringir los valores a un cierto rango, podemos valernos de la función `constrain(valor, min, max);` donde `valor` es el valor a restringir, `min` es el valor a devolver en caso de que `valor` sea menor que `min` y `max` es el valor que debe tomar `valor` si es mayor que `max`. Por ejemplo:

```
int x = 11;
int y = constrain(x,0,10); // y tendrá valor 10
int z = constrain(x,0,100); // z tendrá valor 11 por estar entre los límites
```

Las medidas que se hacen sobre las lecturas analógicas vienen dadas por un rango eléctrico de 0 a 5 voltios por defecto, pero Arduino permite utilizar otros voltajes de referencia que no sean los 5 voltios, para ello existe un pin en la tarjeta marcado como AREF y que significa *Analogue REFerence* o referencia analógica. Con esta entrada podemos indicar a Arduino que utilice otra referencia voltaica de la entrada de la señal a medir. El valor introducido debe ser entre 0 y 5 voltios y debe ser un voltaje constante. La tarjeta Arduino Uno permite tres fuentes de referencia voltaica seleccionables mediante la función `analogReference()`:

- Referencia interna: Se toma como referencia 1.1 voltios. Para seleccionar esta referencia se utiliza la llamada `analogReference(INTERNAL)`.
- Referencia por defecto: Se toma como referencia 5 voltios. Para seleccionar esta referencia se utiliza la llamada `analogReference(DEFAULT)`.
- Referencia externa: Se toma como referencia el valor en la entrada del pin *AREF*, que debe ser entre 0 y 5 voltios. Para seleccionar esta referencia se utiliza la llamada `analogReference(EXTERNAL)`.

Más adelante se utilizará esta función en varios ejemplos.





4 Salidas visuales

En este capítulo aprenderá a:

- Utilizar leds externos.
- Polarizar correctamente un led.
- Controlar salidas PWM.
- Usar pantallas de 7 segmentos.

Hasta el momento hemos descubierto la manera en la que la placa Arduino obtiene información del mundo exterior, tanto de valores analógicos como digitales, pero a la espera de realizar un estudio en mayor profundidad de cómo situaciones físicas externas pueden ser llevadas al microcontrolador a través de sensores, en este tema nos centraremos en conocer un poco más sobre las salidas de estas tarjetas. Por ahora para mostrar información de lo que estaba pasando en el microcontrolador, hemos utilizado bien el monitor serie o bien el led integrado en la placa; va siendo hora de conocer nuevos métodos de mostrar información.
En este capítulo se ahondará en el uso de las salidas y se aprovechará para introducir nuevos elementos a nuestros circuitos: los leds y las pantallas de 7 segmentos.

Led

A lo mejor aún tenemos en mente la pregunta ¿qué es un led? En el sentido estricto led es una abreviatura de *Light Emitting Diode*, que traducido sería diodo emisor de luz; como definición rápida podríamos hablar de que el led es una bombillita pequeña que se utiliza en electrónica y como definición un poco más precisa cada vez hablamos de led hablamos de un componente pasivo que cuando se le aplica una tensión entre sus dos terminales (con la polarización correcta), permite que los electrones lo recorran a la vez que libera energía en forma de fotones, es decir emite luz. La luz emitida no tiene porqué ser visible, podemos encontrar desde leds que emiten en infrarrojo hasta ultravioleta y lógicamente también en el espectro visible. Los más comunes son los rojos y amarillos o anaranjados, aunque actualmente podemos encontrar fácilmente de otros muchos colores como azul o verde. La manera de conseguir distintos colores es variando el material de fabricación. Para aumentar su luminosidad, en su interior se encuentra una cavidad reflectora y su exterior está recubierto por una resina epoxi que hace de lente a la vez que protege el conjunto. Este recubrimiento puede ser tintado del mismo color que la luz que emitirá el diodo o transparente (y aún así emitir en color). En el mercado también se pueden encontrar leds RGB, con 4 patillas y capaces de ofrecer los colores rojo, verde y azul en un mismo dispositivo, otros con tres patillas que emiten en dos colores distintos y algunos cuyo color de emisión es el color blanco.
El led tiene que utilizarse respetando una polaridad, es decir no es como las resistencias que da igual que patilla tenga más tensión; el led tiene dos terminales diferentes denominados ánodo y cátodo y deben conectarse de manera

que el ánodo siempre tenga más tensión que el cátodo; que el borne positivo de nuestra pila esté conectado al ánodo mientras que el negativo al cátodo. Como es tan importante polarizar de manera correcta el led, hay que distinguir entre el ánodo y el cátodo y para ello los led nos ofrecen varias marcas en el componente que denotan el cátodo; 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, pero esto no sirve si se han cortado las patas, por lo que podemos fijarnos en otras características, como que muchos leds tienen en el recubrimiento epoxi 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 truco que se explica al final de este capítulo... o mejor usar otro led, que son muy baratos.

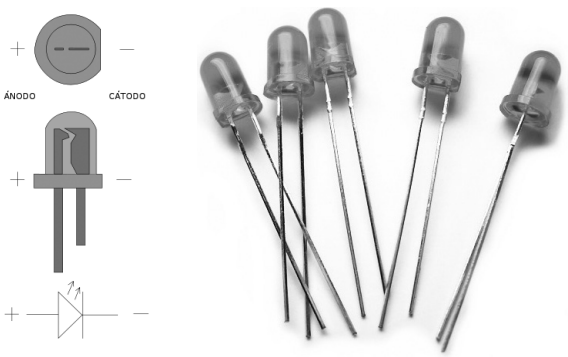


Figura 4.1. Diodos led.

Cuando se hacen los cálculos para el montaje en circuito habría que leerse la hoja de especificaciones del componente, aunque de manera genérica puede decirse que los leds funcionan con una intensidad de 10 a 20 mili amperios dependiendo de si son de baja intensidad o no y respecto a las caídas de tensión depende del color emitido. Para algunos colores podemos tomar como aproximación:

Tabla 4.1. Caídas de tensión en distintos diodos led.

Color	Caída en voltios
Rojo	1.6 - 2.03
Naranja	2.03 - 2.10

Color	Caída en voltios
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

Sabiendo que la salida de Arduino es de 5 voltios y que puede suministrar una corriente de 40mA, debemos de alguna manera limitar la corriente que recorrerá el diodo con tal de no dañarlo. Vamos a echar mano de un poco de fórmulas matemáticas. La ley de Ohm nos dice que V = I * R donde V es el voltaje o diferencia de potencial entre extremos, I es la intensidad que recorrerá el circuito y R es la resistencia entre bornes. Si lo miramos de otra forma, para que esa ecuación se mantenga, si V es constante (que serán los 5V de salida de la placa), cuanto mayor sea la resistencia menor será la intensidad. Lo que haremos será poner una resistencia en serie con el diodo con tal de controlar la corriente que lo recorre y no dañarlo; si nos quedamos cortos de intensidad el led se iluminará de forma tenue, si ponemos poca resistencia podemos dañar irreversiblemente el led fundiéndolo, así que mejor pecar de poner más resistencia de la necesaria (aunque los leds tienen márgenes de seguridad con lo que se pueden usar cálculos aproximados y son muy baratos por lo que la pérdida no sería sustancial).

Usaremos un led rojo, con lo que la caída de tensión sería de unos 2V. Si aplicamos la fórmula para calcular la resistencia teniendo en cuenta que la caída de tensión en la resistencia viene fijada por la disponible en la salida menos la caída que existe en el led, tenemos la ecuación R = (Varduino-Vled)/I =5V-2V/20mA = 150Ω.

Así pues la resistencia a utilizar es como mínimo de 150Ω, pero podemos usar por ejemplo una resistencia de 220Ω que son muy comunes. Para el primer ejemplo utilizaremos simplemente un led conectado a una salida y haremos que parpadee.

En el montaje mostrado en la figura 4.2 se ha alimentado directamente sobre la parte central de la protoboard, esto se ha hecho así porque era un montaje muy sencillo, pero es aconsejable utilizar las líneas de alimentación y tierra situadas en los laterales de la protoboard.

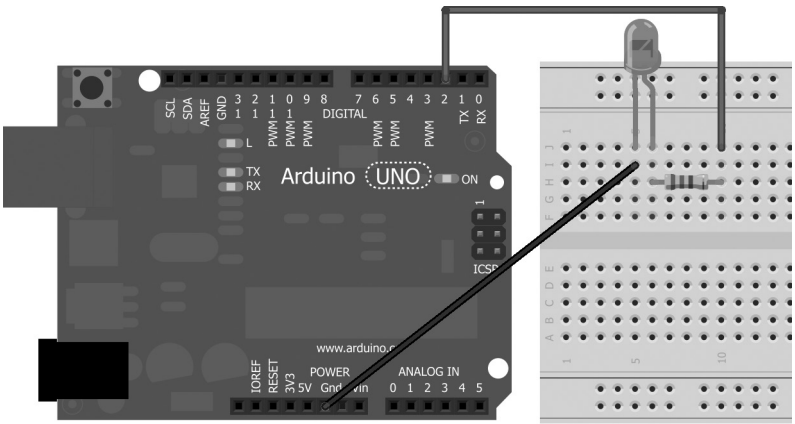


Figura 4.2. Montaje simple con led.

El código para hacer que se ilumine el led es muy simple y prácticamente es idéntico a uno ya utilizado en el capítulo anterior salvo que el pin de salida no es el 13 sino el 2.

```
const int ledPin = 2; // pin del led
int val = 0;

void setup() {
   pinMode(ledPin, OUTPUT);
}

void loop() {
   digitalWrite(ledPin, HIGH); // enciende el led
   delay(1000); // espera 1 segundo
   digitalWrite(ledPin, LOW); // apaga el led
   delay(1000);  // espera 1 segundo
}
```

Semáforo led

Como ya sabemos la manera de conectar un led, vamos a realizar un montaje varios leds. Simularemos un semáforo. En caso de que no se tengan leds de los colores del semáforo, se pueden utilizar tres leds iguales y colocarlos uno encima de otro para simular la posición en el semáforo.
Puesto que este montaje utilizará tres salidas distintas (una para cada led) podemos entonces utilizar directamente la alimentación de la propia placa Arduino, en caso de que el montaje implique varios leds en serie sería necesaria alimentación externa para poder suministrar el voltaje e intensidad necesarios.

El semáforo constará de tres leds, que se irán encendiendo sucesivamente, estando en rojo 9 segundos, en naranja 1 y en verde 8. El ciclo será rojo, verde, naranja, rojo, verde...

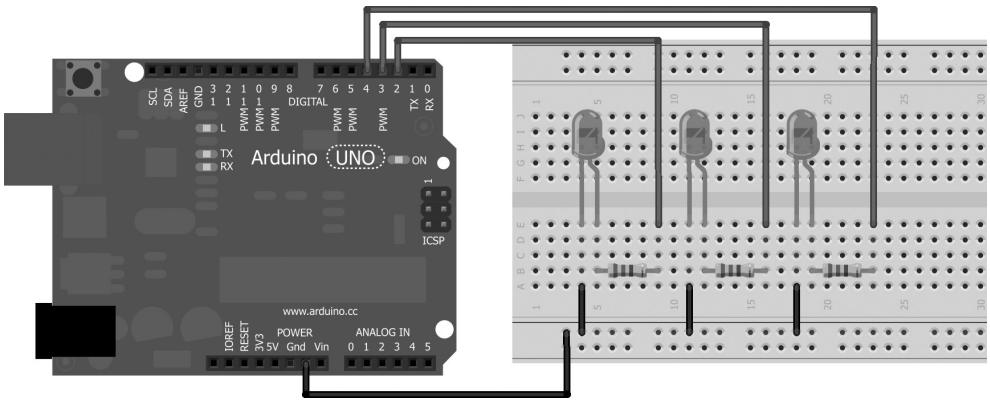


Figura 4.3. Montaje semáforo de leds.

En el circuito todas las resistencias a utilizadas son de 220Ω. En cuanto al código lo primero que se debe hacer es configurar los pines que se utilizarán de salida, tal y como muestra la figura 4.3, los pines de salida serán el 2, 3 y 4 para rojo, naranja y verde respectivamente. En la parte principal del programa se trabajará en tres bloques, uno por cada color; en cada uno de esto bloques se pondrá uno de los leds en funcionamiento a la vez que se apaga el que estuviera encendido, es decir si se enciende el verde se debe apagar el rojo; una vez encendido el led, hacemos un `delay()` correspondiendo con el tiempo que se tiene planificado que se quede encendido.
El listado quedaría:

```
const int redLedPin = 2; // pin del led rojo
const int orangeLedPin = 3; // pin del led naranja
const int greenLedPin = 4; // pin del led verde

void setup() {
   // configuramos los tres pines como salida
   pinMode(redLedPin, OUTPUT);
   pinMode(orangeLedPin, OUTPUT);
   pinMode(greenLedPin, OUTPUT);
}

void loop() {
   // ciclo en rojo
   digitalWrite(orangeLedPin, LOW); // apagamos el naranja
   digitalWrite(redLedPin, HIGH); // encedemos el rojo
   delay(9000); // está en rojo 9 s
```

```
    // ciclo en verde
    digitalWrite(redLedPin, LOW); // apagamos el rojo
    digitalWrite(greenLedPin, HIGH); // encendemos el verde
    delay(8000); // está en verde 8s

    // ciclo en naranja
    digitalWrite(greenLedPin, LOW); // apagamos el verde
    digitalWrite(orangeLedPin, HIGH); // encendemos el naranja
    delay(1000);  // está en naranja 1s
}
```

Este código está bien si simplemente queremos el semáforo sin hacer nada más, pero si se tiene que atender alguna entrada, este código sería totalmente ineficiente. Supongamos que queremos poner un mecanismo por el que se pueda forzar el semáforo a ponerse en verde (en algunas ciudades existen mecanismos así para bomberos, ambulancias y semejantes de modo que no tengan que hacer cola en los semáforos o para peatones que deben pulsar para poder pasar); lo podríamos implementar en nuestro caso mediante un botón, que cuando se apriete el semáforo pase a verde. La primera opción sería poner una lectura en el bucle y comprobar si está pulsado y en tal caso pasar a verde. El problema estriba en que tal y como está escrito el programa, un bucle completo de la función `loop()` tarda 9s+8s+1s es decir 18 segundos. Otra opción sería poner tres lecturas, una detrás de cada ciclo, pero aún y así, si se pulsa mientras está en rojo hay que esperar igualmente a que acabe el ciclo en rojo... es inútil la pulsación. Si se ha programado anteriormente en otros lenguajes a lo mejor vuelve a la cabeza el uso de *threads*, pero no hace falta, si se cambia el código convenientemente se puede trabajar sin ellos.
Pongamos un botón al montaje anterior con entrada en el pin 5 y variemos el código para que sea capaz de atenderlo de inmediato.

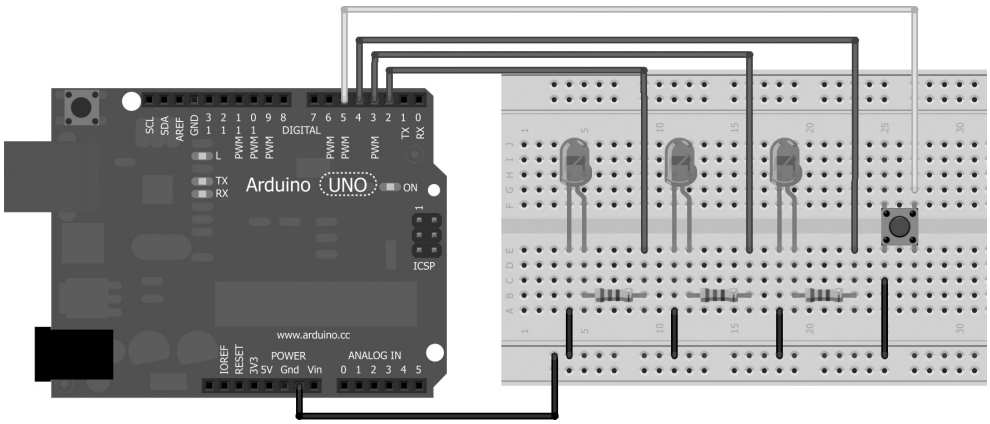


Figura 4.4. Montaje semáforo de leds con pulsador.

Necesitaremos una nueva variable en la que se mantendrá que led se debe encender en cada momento, también realizaremos la definición una constante para el pin de entrada y su consiguiente configuración (activaremos la resistencia de *pull-up* para reducir el número de componentes a utilizar).

```
const int redLedPin = 2; // pin del led rojo
const int orangeLedPin = 3; // pin del led naranja
const int greenLedPin = 4; // pin del led verde
const int buttonPin = 5; // pin del botón
int lightStatus = 0; // led que se debe encender en cada momento
void setup() {
   // configuramos los tres pines de salida
   pinMode(redLedPin, OUTPUT);
   pinMode(orangeLedPin, OUTPUT);
   pinMode(greenLedPin, OUTPUT);
   // Configuramos el botón de entrada
   pinMode(buttonPin, INPUT);
   digitalWrite(buttonPin, HIGH); // se activa la resistencia de pull-up
}
```

En el cuerpo principal leeremos la entrada para ver si está pulsada o no y en caso de estarlo avisar que se tiene que encender el led verde, sea cual sea el que esté encendido. Mediante un `switch` se selecciona el caso en el que nos encontramos dependiendo del led que debe estar encendido y se llama a la función de control de los leds con los parámetros de qué led debe encenderse y el tiempo que debe estar encendido. Hay tres casos posibles, rojo (caso 0), verde (caso 1) y naranja (caso 2), por eso cuando se detecte que el botón está pulsado, forzaremos el `lightStatus` a 1, para inicial el ciclo verde.

```
void loop() {
   if (!digitalRead(buttonPin)){ // si se pulsa
      lightStatus = 1; // se fuerza a verde
   }
   // dependiendo de la luz que se debe mostrar tiene distintos parámetros
   // la llamada
   switch (lightStatus){
      case 0: // red
         showLight(redLedPin,3000);
         break;
      case 1: // green
         showLight(greenLedPin,2000);
         break;
      case 2: // orange
         showLight(orangeLedPin,1000);
         break;
   }
}
```

por último vamos a crear la función `showLight()` a la cual pasaremos como parámetros el led que debe activarse y el tiempo que debe permanecer activo. Dentro de esta función guardaremos el ultimo led que se ha

activado mediante la variable `lastPin` y el tiempo en el que se activó en la variable `startTime`. Al guardar el ultimo led activado podemos compararlo con el led que se tiene que activar en ese momento, si es diferente, se apaga el led activo, se enciende el nuevo y se pone el nuevo como que es el último activado:

```
if (ledPin != lastPin){
   startTime = millis();
   digitalWrite(lastPin, LOW);
   lastPin = ledPin;
   digitalWrite(ledPin, HIGH);
   delay(1);
}
```

Por último se debe comprobar si ya ha pasado el tiempo que el led tenía que estar encendido, y en tal caso pasar al siguiente estado del semáforo:

```
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;   // sólo hay 3 led, comenzamos de
                                          // nuevo por el 0
}
```

Juntando todo, el *sketch* quedaría:

```
const int redLedPin = 2; // pin del led rojo
const int orangeLedPin = 3; // pin del led naranja
const int greenLedPin = 4; // pin del led verde
const int buttonPin = 5; // pin del botón
int lightStatus =0; // led que se debe encender en cada momento

void setup() {
   // configuramos los tres pines de salida
   pinMode(redLedPin, OUTPUT);
   pinMode(orangeLedPin, OUTPUT);
   pinMode(greenLedPin, OUTPUT);

   // Configuramos el botón de entrada
   pinMode(buttonPin, INPUT);
   digitalWrite(buttonPin, HIGH); // se activa la resistencia de pull-up
}

void loop() {
   if (!digitalRead(buttonPin)){ // si se pulsa
      lightStatus = 1; // se fuerza a verde
   }

   // dependiendo de la luz que se debe mostrar tiene distintos parámetros
   // la llamada
   switch (lightStatus){
```

```
      case 0: // red
         showLight(redLedPin,3000);
         break;
      case 1: // green
         showLight(greenLedPin,2000);
         break;
      case 2: // orange
         showLight(orangeLedPin,1000);
         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){
      startTime = millis();
      digitalWrite(lastPin, LOW);
      lastPin = ledPin;
      digitalWrite(ledPin, HIGH);
      delay(1); // 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;   // sólo hay 3 led, comenzamos de
                                             // nuevo por el 0
   }
}
```

Ya tendríamos el semáforo que pasaría a verde nada más pulsar el botón, pero presenta un pequeño problema y es que si ya está en verde el botón no tiene ningún efecto, pero éste podría subsanarse si dentro de la función `loop()`, donde se detecta la pulsación, además de realizar la instrucción `lightStatus = 1;` se iniciara la variable que mantiene el tiempo de inicio de encendido del led `startTime =millis();`, para lo cual tendría que ser global. Supongo que para el lector ya no sería un problema hacer este pequeño cambio y además añadir un segundo semáforo de modo que fuera uno de peatones con pulsador y otro de coches de tal forma que estuvieran sincronizados. ¿Se anima?

Contador binario

Los números en base 2 se representan mediante una serie de 1 y 0, que dependiendo de la posición que ocupen tienen un valor u otro, más concretamente tienen el valor 2^n; por ejemplo en 100, el 1 tiene la posición 2 (las posiciones las contamos desde la derecha y comenzando en 0), así pues su valor es $2^2 = 4$, es decir 100 en binario es 4 en decimal; para 1001, tendríamos $2^3 + 2^0 = 9$. Para este ejemplo usaremos 4 leds para contar de 0 a 15 segundos. Utilizaremos 4 leds para representar 4 bits (o un nibble) pero el ejemplo es fácilmente extensible a 8 bits (un byte) o lo que se quisiera.
El montaje es muy semejante al del semáforo, tan sólo que necesitamos un led más y por lo tanto un pin de salida más. Las resistencias a utilizar serán igualmente de 220Ω.

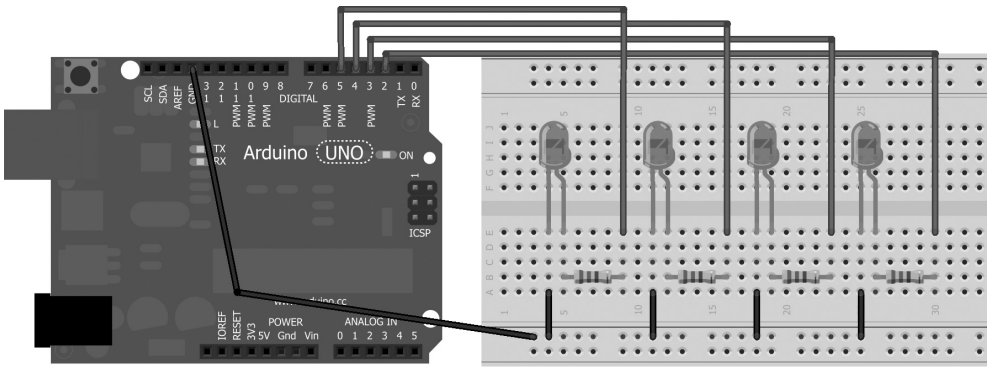


Figura 4.5. Montaje contador binario.

Como son varios los pines de salida a utilizar y están muy relacionados en la lógica del programa, los agruparemos en un array para luego poder trabajar con ellos más fácilmente.

```
const byte ledPin[] = {2,3,4,5}; // Leds a utilizar
```

Usaremos como variables globales la variable `count` que contará de 0 a 15 y servirá para saber el número que se tiene que representar con los leds y la variable `changeTime` que contará cuantos milisegundos lleva transcurridos desde que la variable `count` tomó su valor, o dicho de otro modo cada 1000ms de `changeTime` haremos que cambie en 1 la variable `count`.
La variable `count` podría haberse hecho fácilmente variable local en `loop()`, pero al ser un *sketch* tan sencillo, no recarga la memoria por variables globales y facilita la programación.

Dado que los pines de salida los tenemos en un array podemos hacer la inicialización mediante un bloque `for`, facilitándonos la tarea.

```
for (int x=0; x<4; x++) {
   pinMode(ledPin[x], OUTPUT); //los preparamos para salida
}
```

En el bloque principal llamaremos a la función que se encarga de encender los leds correspondientes cada 1000ms, momento en el que aprovecharemos para inicializar de nuevo el contador de tiempo a la espera que vuelva a llegar a los 1000ms y poder contar un nuevo segundo y pintarlo, y así sucesivamente.

```
if ((millis() - changeTime) > 1000) {  // si ha pasado 1000 milisegundos
                                       // contar uno mas
   count ++;
   setNumber(); // ponemos el numero en los leds
   changeTime = millis(); // inicia el condador de tiempo a 0
}
```

En la función `setNumber()` será donde se realice el encendido y apagado de los leds que hacen la representación de los bits. Lo primero que haremos será controlar que no estemos contando más allá de 16 porque habría desbordamiento; curiosamente, si en este sketch no se controla, visualmente no se notaría, ya que encendería el bit 5 (que no existe en el montaje) y apagaría el resto, con lo que coincide con el 0. Para saber el bit que se debe encender y el que no, hacemos un desplazamiento (*shift*) de los bits del numero a mostrar tantas posiciones como el bit que nos interese saber si está encendido o no y hacemos un *and* con el número 1, de modo que sólo nos quedaremos con el bit situado más a la derecha (el de menor peso). El desplazamiento de bits en Arduino se realiza mediante el operador >> para desplazar a la derecha y << para desplazar a la izquierda, indicando la variable a desplazar y el número de posiciones. Por ejemplo `count << 2` significa que desplace los bits de la variable `count` dos posiciones a la izquierda (los dos bits de más a la izquierda se pierden y por la derecha entran ceros).
Esto se consigue aplicando la siguiente instrucción dentro de un bucle para que analice cada bit.

```
digitalWrite(ledPin[i], (count >> i) & 1);
```

Por ejemplo el número 9 sería 1001, aplicando para el primer bit (comenzando por la derecha) tiene la posición 0, luego desplazamos 0 y la operación 1001 & 0001 = 0001, con lo que el primer led debe estar encendido. En la siguiente iteración veremos el segundo bit (por la derecha), con posición 1; desplazamos el valor de 9 a la derecha una posición, quedando 0100, que al realizar la operación 0100 & 0001 = 0000, es decir el segundo bit no se debe encender, y así sucesivamente.

El código al completo sería:

```
const byte ledPin[] = {2,3,4,5}; // Leds a utilizar
int count=0; // contador
unsigned long changeTime; // contador de tiempo
void setup() {
   for (int x=0; x<4; x++) {
      pinMode(ledPin[x], OUTPUT); // los preparamos para salida
   }
}

void loop() {
   if ((millis() - changeTime) > 1000) {  // si ha pasado 1000 milisegundos
                                          // contar uno mas
      count ++;
      setNumber(); // ponemos el numero en los leds
      changeTime = millis(); // inicia el condador de tiempo a 0
   }
}
/**
* Enciende los leds necesarios para mostrar el numero contenido en la
* variable count
*/
void setNumber (){
   if (count>15) count = 0; // volver a empezar si es mayor de 4 bits

   for (int i = 0; i<4;i++){     // para cada bit mira si tiene que encender
                                 // el led correspondiente
      digitalWrite(ledPin[i], (count >> i) & 1);
   }
}
```

Salida LED con PWM

Hasta el momento siempre hemos hablado de entradas y salidas digitales y entradas analógicas, pero ¿qué pasa con las salidas analógicas? ¿no existen? No y si. No existen como tal pero se simulan mediante salidas digitales pulsadas. El valor analógico es aquel que puede ser variado gradualmente entre su mínimo y máximo, tomando valores intermedios, y mediante la técnica de PWM (*Pulse Width Modulation* o bien modulación por amplitud de pulso) podemos conseguir el efecto de tener un valor analógico usando salidas digitales.

Al utilizar PWM lo que se hace es enviar a la salida pulsos de 0 y 1 muy rápido y dependiendo de la duración del pulso en 1 significarán cantidades distintas, simulando valores entre 0 y 5 voltios. Los valores de escritura en la salida PWM van de 0 a 255. Cuando enviemos una escritura analógica con valor 0, no se realizará ningún pulso en 1, mientras que si se envía de 63 (un cuarto de 255, lo que emularía 1.25V que es un cuarto de 5V), se tendrán

pulsos donde un cuarto del tiempo estará a 1 y el resto a 0, o dicho de otro modo, estaría en HIGH el 25% del tiempo del ciclo. La frecuencia de los pulsos es de 500Hz.

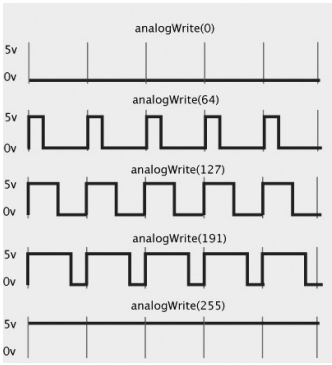


Figura 4.6. Ejemplo PWM.

En la tarjeta tenemos marcados los pines que pueden trabajar con PWM mediante la serigrafía *PWM* o ~, dependiendo de los modelos. Realmente todas las salidas digitales pueden llegar a utilizarse como salida PWM (por si se necesitaran más de 6 salida analógicas), pero es aconsejable utilizar las marcadas para ello. Para demostrar su funcionamiento podemos aprovechar que ya tenemos dominio sobre los leds para seguir jugando con ellos. Utilizaremos un led que se vaya encendiendo y apagando pero en lugar de hacerlo bruscamente como en ejercicios anteriores, lo haremos de manera suave, con un efecto de desvanecimiento. La resistencia usada es de 220Ω.

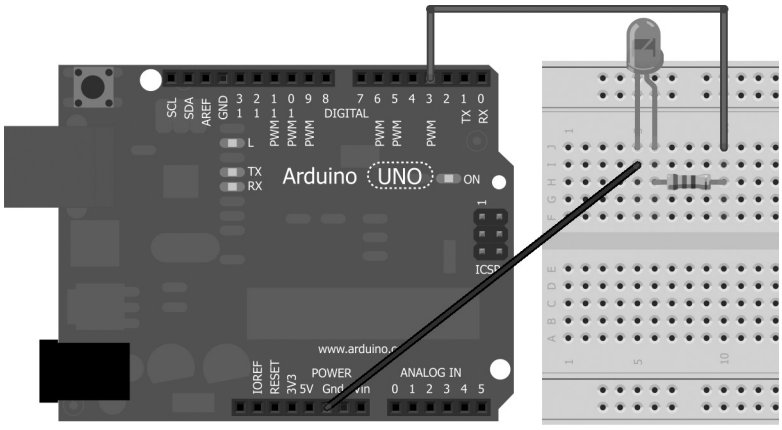


Figura 4.7. Montaje para led pulsante con desvanecimiento.

Para obtener una serie de valores continuos y crecientes decrecientes para hacer el efecto de encendido y apagado suave, echaremos mano de los estudios y rescataremos la función trigonométrica del seno. La función seno daba valores entre 0 y 1 cuando se aplicaba entre 0 y 180 grados, siendo el máximo en 90 grados.

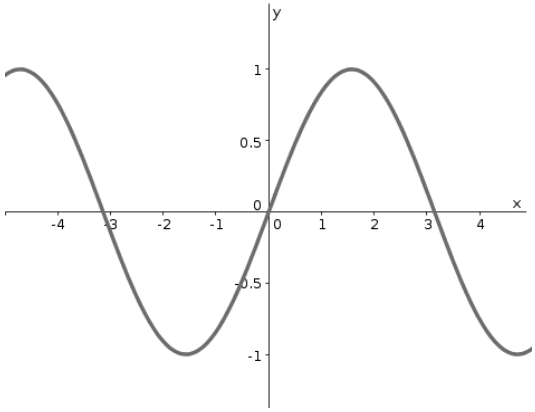


Figura 4.8. Función seno.

En Arduino la función seno viene dada por la función `sin()` (análogamente tenemos `cos()` para el coseno y `tan()` para la tangente), pero no trabaja en grados sino en radianes, por lo que el parámetro hay que transformarlo antes de poder utilizarlo. Para transformar de grados a radianes debemos de multiplicar los grados por ϖ y dividirlo por 180. Si en el bucle principal realizamos el cálculo del seno de 0 a 180 grados iremos consiguiendo valores de 0 a 1 y nuevamente 0, que si lo multiplicamos por el valor máximo que puede tener la salida analógica (que son 255) tendremos el valor a aplicar en cada momento para hacer el efecto deseado. El *sketch* para realizar estas acciones sería:

```
int ledPin = 3; // usamos un pin con PWM
void setup() {
   pinMode(ledPin, OUTPUT);
}

void loop() {
   for (int x=0; x<180; x++) { // sólo los valores positivos del seno
      float sinVal = sin(x*3.1412/180); // calculamos el valor del seno
      int ledVal = int(sinVal*255); // que cantidad de 255 debemos iluminar
      analogWrite(ledPin, ledVal); // salida
      delay(50);
   }
}
```

Modificando el valor de la instrucción `delay(50);` se puede conseguir que el parpadeo vaya más rápido o menos. Si pusiéramos un potenciómetro unido a una de las entradas podríamos leer el valor de éste y tener el ritmo de la pulsación controlado desde el mundo físico.

Obtención de efectos RGB con PWM

En el ejercicio anterior hemos visto como realizar un desvanecimiento de luz en un led (que seguramente habremos elegido color rojo por ser el más habitual). Vamos a hacerlo un poco más divertido y en lugar de realizarlo con un solo led vamos a trabajar con tres leds uno rojo, uno verde y otro azul. Efectivamente, con estos tres colores podemos simular la manera en la que muchas pantallas crean las imágenes mediante sus puntos, a través de la suma de los tres colores es posible crear toda la variedad de colores; para trabajar con RGB (Red Green Blue) lo que se hace es dar un valor a cada uno de los colores y la suma de ellos da el valor esperado. La cantidad de cada valor puede darse de muchas maneras, por ejemplo mediante tantos por ciento, en ese caso 100%,100%,0 seria todo el rojo posible, todo el verde posible y nada de azul y así obtendríamos el amarillo (no hay que confundirse con lo que nos enseñaron de pequeños para crear los colores a partir de los primarios rojo amarillo y azul). La representación con la que trabajaremos nosotros es ajustándonos a los posibles valores que puede tener la salida digital, de 0 a 255, es decir trabajando en 8bits, lo cual da un total de 255*255*255 combinaciones posibles que son más de 16 millones de colores, más de lo que el ojo humano puede percibir (y más dependiendo del sexo del observador según cuentan las malas lenguas). Así el rojo sería 255,0, 0 y el 15,162,153 daría un azul verdoso.

Mediante los valores que vayamos dando en la salida de cada uno de los leds se obtendrá un tono distinto de cada uno de ellos y la suma de todos generará un tercer color dentro de esos 16 millones. Si hacemos que vayan variando de poco en poco cada uno de los tres valores de modo independiente, conseguiremos ir creando transiciones suaves entre todos los colores. Este será el resultado que se espera conseguir con el siguiente ejemplo. Lo que se va a hacer es tener una tripleta de valores RGB a los cuales queremos llegar en la salida de cada uno de los leds, esta tripleta será aleatoria; como cada uno de los leds tendrá ya un valor (0 al iniciar la ejecución), el incremento (positivo o negativo) a realizar será `((valor_esperado) - (valor_actual)/número_de_iteraciones))`, y con esto obtendríamos el incremento a sumar a cada uno de los valores en cada iteración, de modo que después de

haber ejecutado `número_de_iteraciones` veces, tendremos que la tripleta de valores actuales y deseados coincide, entonces generaremos aleatoriamente una nueva tripleta de valores destino y los incrementos correspondientes para alcanzar los valores en un `número_de_iteraciones` y así sucesivamente con tal de alcanzar esas transiciones aleatorias y suaves.
El montaje se ha dispuesto en la protoboard de manera que resulte sencillo de seguir, pero el mejor efecto de este circuito se consigue cuanto más cerca se encuentren los tres leds, por lo que animo al lector a modificar el circuito (siempre respetando las polaridades de los leds) añadiendo un par de cables para poder aislar los leds y ponerlos juntos. También recomiendo al lector que utilice los leds más pequeños y potentes que encuentre para mejorar el efecto.

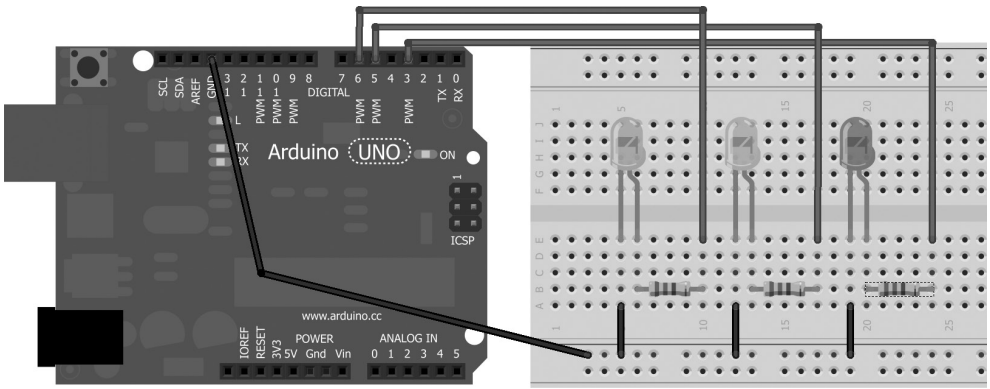


Figura 4.9. Circuito para unión de leds PWM.

En el *sketch* de control del circuito crearemos tres arrays de tres valores cada uno de tipo `float` que mantendrán los valores actuales de cada uno de los leds de salida, los valores finales que deben alcanzar y el incremento que se debe hacer en cada iteración con tal de que el valor actual llegue al valor final. Estos arrays estarán en orden RGB, es decir el primer elemento correspondería al color rojo y el último al azul. En la función de `setup()` se procede a la inicialización de los números aleatorios. Como es costumbre en los sistemas electrónicos, la mejor manera de obtener números aleatorios (realmente pseudo aleatorios) es mediante una semilla lo más aleatoria posible. En Arduino se suele utilizar una lectura a un pin analógico que no tenga nada unido a él, es decir captará el valor correspondiente al ruido eléctrico que haya en la entrada que puede verse influenciado por múltiples factores, como lo cerca que lo tengamos de un enchufe o si está sonando un teléfono en las proximidades. También se da valores a los arrays de "salidas actuales" y "salidas destino"; las salidas actuales serán a 0 y las esperadas ya

serán aleatorias. Al comenzar en 0, nuestros leds empezarán apagados y se irán encendiendo hasta el valor obtenido en el cálculo del número aleatorio correspondiente a su valor destino.

```
float RGBfrom[3]; // valores actuales
float RGBto[3]; // valores destino
float increment[3]; // incrementos
// pines de salida pwm
int redPin = 3;
int greenPin = 5;
int bluePin = 6;
void setup(){
   pinMode(redPin, OUTPUT);
   pinMode(redPin, OUTPUT);
   pinMode(redPin, OUTPUT);
   pinMode(A0, INPUT);
   randomSeed(analogRead(A0)); // inicializa el random
   // comienzo en 0
   RGBfrom[0] = 0;
   RGBfrom[1] = 0;
   RGBfrom[2] = 0;
   // valores destino aleatorios
   RGBto[0] = random(256);
   RGBto[1] = random(256);
   RGBto[2] = random(256);
}
```

El el bucle principal calcularemos los incrementos a realizar en cada iteración sabiendo que se realizarán en 256 iteraciones. Se ha elegido este número de iteraciones porque lo más desfavorable es estar con valor actual en 0 o 255 y que como valor esperado se obtenga 255 o 0 respectivamente, y esto son 256 estados distintos; si se está más cerca, pues el incremento en cada iteración será menor que uno. Tras el cálculo de los incrementos llamaremos a una función que se encargue de escribir los valores correspondientes en la salida hasta que se consigan los valores esperados en la salida. Una vez acabada la ejecución de la función, o sea cuando se hayan obtenido en la salida los valores obtenidos aleatoriamente, crearemos unos nuevos valores aleatorios a los que dirigir los valores de salida, y comenzaremos un nuevo ciclo.

```
void loop(){
   // cálculo de incrementos a sumar en cada iteración
   for (int x=0; x<3; x++) {
      increment[x] = (RGBfrom[x] - RGBto[x]) / 256;
   }
   // realizamos las salidas para los cálculos
   setOutput();
   // se han alcanzado los valores destino
   // generamos nuevos valores destino
   for (int x=0; x<3; x++) {
      RGBto[x] = random(256);
   }
}
```

La función `setOutput()` se encarga de realizar las iteraciones escribiendo en la salida PWM el valor que corresponda y calcular el nuevo valor a mostrar dependiendo de los incrementos para cada uno de los colores RGB.

```
/***
* Escribe en la salida el valor actual de cada led
* y ajusta el nuevo valor según el incremento calculado
***/
void setOutput(){
   // 256 iteraciones para llegar al valor destino
   for (int x=0; x<256; x++) {
      analogWrite (redPin, int(RGBfrom[0]));
      analogWrite (greenPin, int(RGBfrom[1]));
      analogWrite (bluePin, int(RGBfrom[2]));
      delay(10); // ajustable según deseo
      // se ajusta el valor actual del led, según el incremento
      // correspondiente a cada valor
      for (int x=0; x<3; x++) {
         RGBfrom[x] -= increment[x];
      }
   }
}
```

Si el número de iteraciones utilizadas nos parecen excesivas para obtener el valor destino, se pueden disminuir, lo que hará que las transiciones sean más rápidas pero a la vez más bruscas, y si se rebaja mucho el numero de iteraciones puede que veamos algún pequeño salto de color. Para modificar algo la velocidad se ha incluido la instrucción `delay(10);` que se encarga de añadir una pequeña pausa entre cambios de valor en la salida, aumentando el valor pasado a la función disminuiremos la velocidad de cambio. Si este valor viene dado por una entrada externa podríamos controlar la velocidad de cambio, por ejemplo mediante un potenciómetro.

```
float RGBfrom[3]; // valores actuales
float RGBto[3]; // valores destino
float increment[3]; // incrementos
// pines de salida pwm
int redPin = 3;
int greenPin = 5;
int bluePin = 6;
void setup(){
   pinMode(redPin, OUTPUT);
   pinMode(redPin, OUTPUT);
   pinMode(redPin, OUTPUT);
   pinMode(A0, INPUT);
   randomSeed(analogRead(A0)); // inicializa el random

   // comienzo en 0
   RGBfrom[0] = 0;
   RGBfrom[1] = 0;
   RGBfrom[2] = 0;
```

```
   // valores destino aleatorios
   RGBto[0] = random(256);
   RGBto[1] = random(256);
   RGBto[2] = random(256);
}

void loop(){
   // cálculo de incrementos a sumar en cada iteración
   for (int x=0; x<3; x++) {
      increment[x] = (RGBfrom[x] - RGBto[x]) / 256;
   }
   // realizamos las salidas para los cálculos
   setOutput();
   // se han alcanzado los valores destino
   // generamos nuevos valores destino
   for (int x=0; x<3; x++) {
      RGBto[x] = random(256);
   }
}

/***
* Escribe en la salida el valor actual de cada led
* y ajusta el nuevo valor según el incremento calculado
***/
void setOutput(){
   // 256 iteraciones para llegar al valor destino
   for (int x=0; x<256; x++) {
      analogWrite (redPin, int(RGBfrom[0]));
      analogWrite (greenPin, int(RGBfrom[1]));
      analogWrite (bluePin, int(RGBfrom[2]));
      delay(10); // ajustable según deseo
      // se ajusta el valor actual del led, según el incremento
      // correspondiente a cada valor
      for (int x=0; x<3; x++) {
         RGBfrom[x] -= increment[x];
      }
   }
}
```

Pantallas de 7 segmentos

Las pantallas o *displays* de 7 segmentos son unos dispositivos que sirven para mostrar números y que están ampliamente extendidos por su facilidad de uso frente a otras soluciones como pueden ser los indicadores o displays de puntos.

Estas pantallas constan internamente de 7 leds que externamente se presentan en forma de barras convenientemente colocadas llamadas segmentos. Dependiendo de los segmentos que se iluminen, formarán los números del 0 al 9.

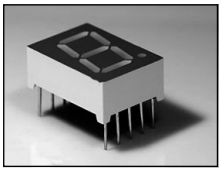


Figura 4.10. Display de 7 segmentos.

Cada uno de los segmentos que conforman la pantalla son conocidas por una letra tal y como se muestra en la figura 4.11, y esta nomenclatura se utiliza de modo estándar sea cual sea el fabricante, es decir siempre que se refieran al segmento A será el segmento superior (no es estándar la numeración de las patillas). Se presentan de dos tipos de ánodo común y de cátodo común. Los de cátodo común, como su propio nombre indica, todos los leds tienen unidos internamente sus cátodos. La activación de todos ellos se realiza aplicando niveles lógicos 1 a sus entradas (5 voltios) y tierra al cátodo común. En los de ánodo común, la unión interna corresponde a todos los ánodos y la activación se realiza aplicando niveles lógicos 0 a sus entradas, siendo la patilla común conectada a la alimentación. Así pues si se hace recuento se necesitan 7 patillas para cada uno de los segmentos y una adicional que corresponde al cátodo o ánodo común de todos ellos; un total de 8 patillas. Estos encapsulados normalmente vienen con más patillas y se aprovecha para introducir elementos adicionales como el punto decimal o patillas comunes extras.

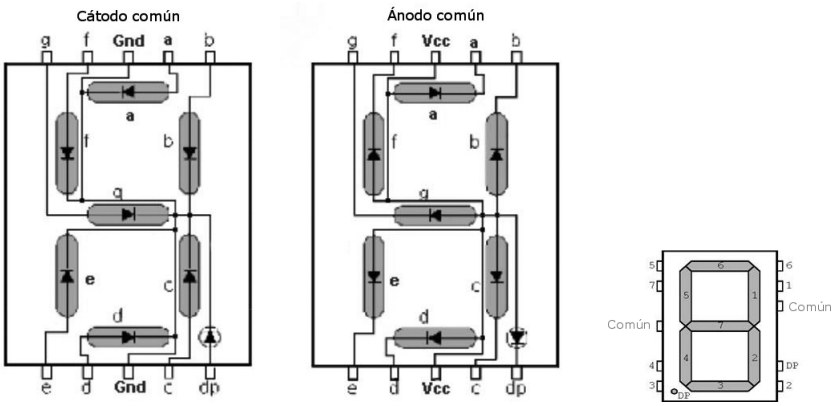


Figura 4.11. Conexiones internas y numeración de patillas.

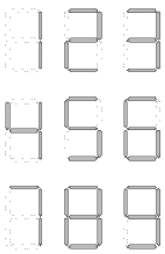
Para distinguir las patillas del display se utilizan dos convenciones, la de la letra del segmento que ilumina (y DP para el *Digital Point* o punto digital) o mediante un número de pin; pines que **normalmente** se comienzan a contar

desde abajo a la izquierda, de izquierda a derecha y se continua por arriba de derecha a izquierda en el caso de que tenga los pines arriba y abajo, y en caso de que los tenga en los laterales se cuentan de arriba a abajo empezando por la izquierda y continuando por la derecha de abajo a arriba.

Advertencia:

Siempre es recomendable revisar las hojas del fabricante o la serigrafía de las patillas del componente para conocer qué patilla corresponde a cada segmento.

Como el encapsulado son simplemente leds, ya sabemos que tienen polaridad y hay que respetarla y también que hay que limitar la corriente que circula por ellos, utilizando como con los leds convencionales resistencias limitadoras de corriente. Para los ejercicios que vamos a realizar utilizaremos resistencias de 220Ω que nos valdrán para la mayor parte de los displays (ya que la caída es cerca de 2V en cada led y la corriente admitida entre 10 mA y 20 mA) aunque si es un circuito que va a estar mucho tiempo encendido es mejor leer las hojas del fabricante para asegurarnos de las resistencias que se deben utilizar. Normalmente este tipo de displays se utiliza conjuntamente con otros chips que ayudan a mostrar el número en pantalla, como los decodificadores BCD/7 segmentos que simplemente introduciendo el dato a mostrar en código BCD (para saber más sobre el código BCD ver los apéndices) él ya enciende los segmentos que corresponden. Nosotros no vamos a trabajar con este tipo de codificadores sino que accederemos directamente a encender uno a uno los leds. Con el fin de conseguir iluminar los diferentes números nos podemos guiar de la figura 4.12.



Dígito	Estado del segmento						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	0	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Figura 4.12. Tabla de iluminación de display 7 segmentos.

Para comprender mejor su funcionamiento vamos a crear un *sketch* que cuente de 0 a 9 y lo muestre en el display de 7 segmentos a la vez que enciende el punto decimal uno de cada dos segundos.

Para el montaje necesitaremos resistencias de 220Ω y lógicamente un display de 7 segmentos que en nuestro caso será de cátodo común.

Advertencia:

Si se dispone de ánodo común se debe modificar el circuito para que el común vaya a alimentación y el sketch *para acomodarlo a que trabaje con valores de activación* LOW.

Como son leds independientes, así los tenemos que "atacar" y necesitaremos realizar una conexión nueva por cada led que queramos encender y el control de cada uno de ellos también lo realizaremos mediante pines distintos de salida. Para no complicar el circuito con excesivos cables y se pueda ver con claridad las entradas conectadas, sólo se ha puesto una resistencia en el común de los cátodos, pero este montaje se suele realizar con resistencias separadas para cada una de las patillas con tal de tener mejor fiabilidad.
Para que aún quede más clara la conexión entre los pines del display y los de Arduino teniendo en cuenta que la numeración entre fabricantes puede variar, se adjunta la tabla 4.2.

Tabla 4.2. Conexiones entre Arduino y el display de 7 segmentos.

Pin Arduino	Segmento
2	A
3	B
4	C
5	D
6	E
7	F
8	G
9	DP

En cuanto al *sketch* se refiere, para ayudarnos a la hora de pintar los números vamos a crear un array de 10 posiciones conteniendo cada una de ellas un array de 7 posiciones con los estados de los leds, de este modo podremos recorrerlo mediante un bucle y ahorraremos mucho código.

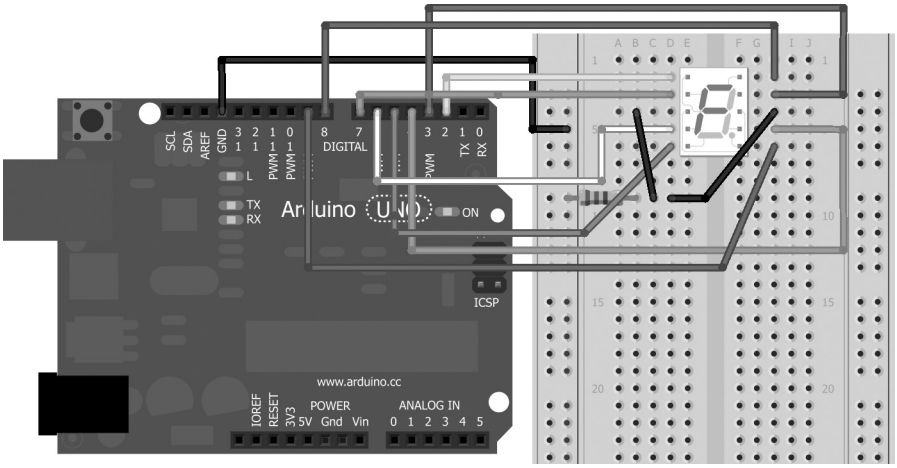


Figura 4.13. Conexiones para circuito con display de 7 segmentos.

Por ejemplo para poner el numero 5 necesitaríamos encendidos los segmentos A, C, D, F, G estando apagados el B y el E, así pues lo podemos representar por { 1,0,1,1,0,1,1 }, donde cada entrada es uno de los segmentos del A al E.

```
const byte sevenSegDigits[10][7] = { { 1,1,1,1,1,1,0 },  //  0
      { 0,1,1,0,0,0,0 },  //  1
      { 1,1,0,1,1,0,1 },  //  2
      { 1,1,1,1,0,0,1 },  //  3
      { 0,1,1,0,0,1,1 },  //  4
      { 1,0,1,1,0,1,1 },  //  5
      { 1,0,1,1,1,1,1 },  //  6
      { 1,1,1,0,0,0,0 },  //  7
      { 1,1,1,1,1,1,1 },  //  8
      { 1,1,1,0,0,1,1 }   //  9
      };
boolean showDot ;
```

En la parte correspondiente a la función `setup()` configuramos como salida los pines del 2 al 8, utiliaremos del 2 al 8 para los 7 segmentos y el 9 para el punto.
En el `loop` principal nos encargaremos de contar de 0 a 10 e ir escribiendo en el *display* mediante la función `sevenSegWrite()` y mostrando el punto mediante la función `writeDot()`.

```
for (byte number = 0; number <  10; number++) {
   writeDot(showDot); // escribir el punto
   showDot =!showDot; // cambiar de estado para el siguiente ciclo
   sevenSegWrite(number); // escribir el número
   delay(1000); // esperar  1s
}
delay(2000); // fin de conteo, esperar 2s antes de comenzar de nuevo
```

A lo mejor el lector ha caído en la cuenta de que estamos utilizando byte en lugar de `int` para los bucles. Esto es porque el bucle va a llegar simplemente hasta 10 y este valor se puede representar en un `byte` y así ahorramos memoria ya que un `int` ocupa 2 `bytes`. Esto se ha de tener muy en cuenta si se trabaja con aplicaciones cuyo manejo de memoria es crítico, pero en pequeñas aplicaciones no es importante y pueden usarse indistintamente. La función `writeDot()` simplemente tiene que escribir en la salida el valor obtenido como parámetro.

```
void writeDot(boolean dot) {
   digitalWrite(9, dot);
}
```

La función `sevenSegWrite()` es la que tiene toda la lógica de la escritura y vamos a aprovechar alguna de las funcionalidades del bloque `for` para hacer el código más compacto. Lo que debe de hacer es obtener el array del número que se le pase como parámetro y recorrer cada uno de sus 7 valores escribiendo en el pin correspondiente el valor que tenga almacenado en el array. Sabemos que el pin de salida es el 2 y que almacena el segmento A y que los pines están de manera consecutiva junto a los segmentos, es decir en el pin 3 está el segmento B, en el 4 el C... de modo que a la vez que recorremos los 7 valores del array podemos ir recorriendo los 7 pines, teniendo en cuenta que los pines comienzan en el índice 2 y los valores en el índice 0. No debemos olvidar incrementar los índices tanto del pin de escritura como del segmento que se está leyendo del array. Todo esto traducido a código sería:

```
void sevenSegWrite(byte digit) {
   for (byte segCount = 0, pin=2; segCount < 7; segCount++, pin++) {
      digitalWrite(pin, sevenSegDigits[digit][segCount]);
   }
}
```

Otra opción, podría haber sido utilizar solamente un índice pero a la hora de realizar el acceso al array y los pines tener en cuenta este desfase de 2 entre ellos, por ejemplo:

```
digitalWrite(index+2, sevenSegDigits[digit][index]);
```

El listado completo del *sketch* sería:

```
const byte sevenSegDigits[10][7] = { { 1,1,1,1,1,1,0 },  //  0
      { 0,1,1,0,0,0,0 },  //  1
      { 1,1,0,1,1,0,1 },  //  2
      { 1,1,1,1,0,0,1 },  //  3
      { 0,1,1,0,0,1,1 },  //  4
      { 1,0,1,1,0,1,1 },  //  5
      { 1,0,1,1,1,1,1 },  //  6
      { 1,1,1,0,0,0,0 },  //  7
      { 1,1,1,1,1,1,1 },  //  8
```

```
      { 1,1,1,0,0,1,1 }   //  9
      };
boolean showDot ;

void setup() {
   for (byte i = 2; i < 10; i++){  // ponemos los pines de salida
   pinMode(i, OUTPUT);
   }
   showDot = false; // comenzamos con el punto apagado
}
void loop() {
   for (byte number = 0; number <  10; number++) {
      writeDot(showDot); // escribir el punto
      showDot =!showDot; // cambiar de estado para el siguiente ciclo
      sevenSegWrite(number); // escribir el número
      delay(1000); // esperar  1s
   }
   delay(2000); // fin de conteo, esperar 2s antes de comenzar de nuevo
}

/***
* Muestra el punto decimal dependiendo del parámetro
***/
void writeDot(boolean dot) {
   digitalWrite(9, dot);
}
/***
* Escribe el número pasado como parámetro en el display
***/
void sevenSegWrite(byte digit) {
   // Recorre el array definido como constante del número pasado como
   // parámetro
   for (byte segCount = 0, pin=2; segCount < 7; segCount++, pin++) {
      digitalWrite(pin, sevenSegDigits[digit][segCount]);
   }
}
```

Para hacernos una idea del ahorro obtenido al utilizar el array con toda la definición de los leds de cada número, si no lo hubiéramos usado, en el *sketch* cada vez que se tuviera que pintar un número habría que escribir la salida de los 7 segmentos que componen dicho el número a mano, poniendo a `LOW` los segmentos apagados y a `HIGH` los encendidos de éste modo por ejemplo para escribir el 6 sería:

```
// escribir  6
digitalWrite(2, 1);
digitalWrite(3, 0);
digitalWrite(4, 1);
digitalWrite(5, 1);
digitalWrite(6, 1);
digitalWrite(7, 1);
digitalWrite(8, 1);
```

Y esto con cada uno de los 10 números.

Displays 7 segmentos con múltiples dígitos

Cuando se quieren mostrar números de más de 1 dígito se puede optar por realizar el montaje creado en el ejercicio anterior para cada número o utilizar displays de 7 segmentos con múltiples dígitos (o usar otro tipo de pantallas, está claro). Los más comunes son de 4 dígitos y se pueden encontrar tanto de ánodo común como de cátodo común. El problema es que si para 1 dígito teníamos 8 pines como mínimo, para 4 dígitos serían 32 pines, lo cual sería complicado de cablear y fácil equivocarse.

Figura 4.14. Display de 7 segmentos con múltiples dígitos.

Lo que se hace es que estos componentes tienen los 7 pines de cada uno de los segmentos (como los tenían los de un solo dígito) comunes a todos los dígitos y el común (ánodo o cátodo) es propio de cada uno de los dígitos.
Dicho de otra manera, cuando se activa el segmento B, se activa para todos los dígitos y es mediante el común de cada uno de los dígitos mediante el que se controla cual es el que se debe encender, supongamos que estamos con un componente de cátodo común, pues al activar los segmentos se activarán en todos pero lucirán tan sólo aquellos que tengan el cátodo a masa (para que circule la corriente por el led). Así tendríamos 7 pines para los segmentos, uno para el dígito de control y 4 para los cátodos, con lo que tenemos los 12 pines con los que vienen equipados estos componentes.
Entonces... ¿Cómo se hace para que muestren diferentes números en cada uno de los dígitos? Pues engañando al ojo. Lo que se hace es mostrar uno de los dígitos, luego otro, luego otro y cuando se acaban de mostrar los dígitos uno a uno, se vuelve a comenzar y es nuestro ojo quien se encarga de realizar la magia para que parezcan que están todos encendidos. En resumen, cuando queremos pintar el número 425, deberíamos proceder de la siguiente manera (en caso de tener un cátodo común):

- Mostrar el primer dígito.
  - Activar los segmentos B, C, F y G.
  - Poner a 0 el pin correspondiente al cátodo del primer dígito.

    - Esperar unos milisegundos.
    - Poner a alta impedancia o bien a 1 el pin correspondiente al cátodo del primer dígito.
- Mostrar el segundo dígito.
    - Activar los segmentos A, B, D, E y G.
    - Poner a 0 el pin correspondiente al cátodo del segundo dígito.
    - Esperar unos milisegundos.
    - Poner a alta impedancia o bien a 1 el pin correspondiente al cátodo del segundo dígito.
- Mostrar el segundo dígito.
    - Activar los segmentos A, C, D, F y G.
    - Poner a 0 el pin correspondiente al cátodo del tercer dígito.
    - Esperar unos milisegundos.
    - Poner a alta impedancia o bien a 1 el pin correspondiente al cátodo del tercer dígito.

El orden en el que se muestren los dígitos da igual, es lo mismo encender de derecha a izquierda, de izquierda a derecha o bien como se quiera, puesto que para el ojo humano estarán todos encendidos. El tiempo de espera sirve para que la retina guarde la información sobre el número que está encendido y debe ser suficiente para que se grabe la imagen en ella pero no muy alto porque entonces cuando mostremos el cuarto número el primero habrá desaparecido de nuestra cabeza, haciendo un efecto de parpadeo (*flickering*). La retina guarda las imágenes una décima de segundo, por lo que el tiempo que se suele utilizar en estos dispositivos es de 30ms por dígito.
La forma de trabajar desde el sketch sería semejante al del ejemplo del display de 7 segmentos de un sólo dígito.
Crearíamos las constantes con los segmentos y la función `setNumber()` tendría dos parámetros, uno el número a mostrar y otro con la posición en la que se quiere mostrar.

```
void setNumber (byte nPosition, byte digit){
   for (byte i = 0; i < 4; i ++){
      pinMode(cathodes[i], INPUT);
   }
   pinMode(cathodes[nPosition], OUTPUT);
   digitalWrite(cathodes[nPosition], LOW);
   for (byte segCount = 0, pin=2; segCount < 7; segCount++, pin++) {
      digitalWrite(pin, sevenSegDigits[digit][segCount]);
   }
}
```

Lo primero que hacemos es poner todos los cátodos en alta impedancia para que no muestren el dígito, en el array `cathodes` tendríamos los pines de control de los cátodos. Luego pondríamos a 0 el cátodo del que nos interese mostrar, obteniendo así "paso" para los electrones en los leds que se activen. Por último escribiríamos los segmentos como lo hicimos con el display de 7 segmentos en el ejercicio anterior.
Dentro del `loop` principal para obtener por ejemplo el 0425 escribiríamos algo semejante a:

```
void  loop (){
   setNumber (0,5);
   delay(30);
   setNumber (1,2);
   delay(30);
   setNumber (2,4);
   delay(30);
   setNumber (3,0);
   delay(30);
}
```

y en caso de no querer que apareciera el 0, simplemente no se informa nada en ese dígito. La llamada a la función `delay()` también se puede incluir dentro de la función `setNumber()` si se quiere que todos los dígitos permanezcan el mismo tiempo encendidos como es en este caso.
Antes de terminar el tema de los leds, me gustaría explicar un par de trucos sobre ellos.
El primero es sobre es detectar la polaridad, aunque ya hemos explicado unas cuantas técnicas para ver en un diodo cuál terminales es el ánodo y cuál es el cátodo, estas técnicas no sirven para ver si un display de 7 segmentos es de ánodo común o cátodo común ya que todas las patillas son iguales y no tenemos forma de ver el diodo físicamente por dentro. Lo que se puede hacer en estos casos es utilizar un multímetro como si se fuera a medir una resistencia. Cuando en el multímetro se selecciona la opción de medir resistencias, se genera una pequeña diferencia de potencial entre los bornes del multímetro, que podemos aplicar directamente a las patas del display. Poniendo uno de los bornes del multímetro sobre una de las patas "común" del display y el otro borne sobre otra pata (que no sea "comun") pueden pasar dos cosas, que se ilumine algún segmento, que indicará que estamos en ese momento en polarización directa o que no se encienda nada que puede ser que estemos en polarización inversa, que estemos poniendo mal los bornes, que el display está fundido o que el multímetro no tiene pilas. Por ejemplo si ponemos en un display de cátodo común el borne positivo (rojo) del multimetro en la patilla del segmento E (normalmente la 1) y el borne negativo (negro) en la patilla común (normalmente la 3 y 8) entonces se encenderá el

segmento E, mientras que si ponemos los bornes al revés no se encenderá nada y no fundiremos el led porque la intensidad que lo recorre es muy baja. Esta técnica lógicamente vale también para los leds individualmente. Cuando se realizan estas comprobaciones, es aconsejable utilizar escalas pequeñas de medición de resistencias en el multímetro.

El segundo truco tiene que ver también con la polaridad y versa sobre la protección que podemos poner a un led frente a una polaridad inversa. Supongamos que hemos realizado un circuito con todo el cuidado del mundo y hemos polarizado perfectamente los bornes positivo y negativo de manera que el usuario cuando lo vaya a utilizar simplemente tenga que conectar las baterías según la polaridad que marca la serigrafía en cada borne. Pero el usuario en un despiste conecta el polo negativo de la batería al borne positivo de nuestro circuito y el polo positivo al borne negativo. En esos momento tendríamos los led (y el resto de componentes) polarizados a la inversa.

Hay componentes que la polarización no les afecta, tales como las resistencias o las inductancias, pero otros como los leds o los condensadores electrolíticos si se ven afectados y se les debe proteger. Para la protección utilizaríamos un componente llamado diodo (el led es un tipo de diodo). El diodo tiene la propiedad de que idealmente cuando se polariza de manera directa se comporta como un interruptor cerrado, con resistencia 0 (en la realidad tiene un poco de resistencia) y si se polariza de modo inverso se comporta como un interruptor abierto, con resistencia ∞ (realmente conduce muy muy poca corriente). Sabiendo esto podemos configurar unos diodos en serie con el led de modo que cuando se polariza al revés idealmente no circularía la corriente (prácticamente 0). Los diodos tienen marcado su ánodo mediante una raya en uno de sus extremos.

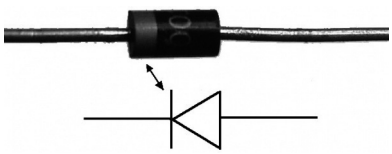


Figura 4.15. Diodo.

5 Conexiones serie

En este capítulo aprenderá a:

- Manejar entradas y salidas del monitor serie.
- Manejar los pines de comunicación.
- Comunicarse con programas de ordenador.
- Realizar comunicaciones entre tarjetas Arduino.

Aunque a lo mejor no nos hayamos dado cuenta, desde el primer momento que hemos cargado un *sketch* en la tarjeta venimos utilizando la comunicación serie, dado que la transferencia de los programas se realiza utilizando este tipo de comunicación; esto lo podemos ver fijándonos en la tarjeta justo en el momento de la carga del *sketch*, en la que durante unos instantes habrá unos leds que parpadearán, esto significa que se está produciendo una transmisión, pero aquí no intervenimos, es totalmente transparente para nosotros. Una comunicación serie menos transparente para nosotros es la que se realiza mediante el monitor serie que ya hemos utilizado brevemente en algún que otro ejercicio. En este capítulo aprenderemos a sacarle mayor partido a este monitor y avanzaremos hacia las comunicaciones con el mundo exterior tanto con programas de ordenador como con otras tarjetas Arduino.

Dentro de la comunicación serie intervienen estos 2 factores: el software y el hardware. El software tiene el control de qué es lo que se va a enviar y el hardware lo tiene sobre el cómo lo va a enviar. El software se encargará de, mediante ciertas llamadas, indicar al hardware los bits que tiene que hacer llegar a destino. En nuestro caso, Arduino ofrece librerías para aislarnos de la complejidad de trabajar con el hardware directamente. En cuanto a la misión del hardware, es enviar y recibir pulsos eléctricos que conformarán los datos transmitidos. El hardware involucrado en la comunicación debe ser compatible entre sí, ya que para algunos sistemas el lógico son 0V, para otros 5V y para otros pueden ser 12V o cualquier otro valor. En el caso de Arduino el 1 son 5V (3.3V en algunos modelos) y el 0 son 0V que se suelen denominar niveles TTL (*Transistor Transistor Logic*, lógica transistor transistor) frente a otros niveles por ejemplo el DTL (*Diode Transistor Logic*, lógica diodo transistor) en el que el 0 lógico está a 1V.

Es posible que al pensar en comunicaciones serie a alguno le venga a la cabeza un viejo conocido, el conector RS-232. Esta conexión estaba presente en los ordenadores antiguos, se trataba de un conector en forma de trapecio con 9 pines llamado DB9 y servía principalmente (aunque no exclusivamente) para la comunicación con el ratón. Actualmente pocos ordenadores lo tienen y para aquellos que lo tengan y lo pensaran usar, hay que decir que el RS-232 no es compatible con Arduino, ya que este protocolo acepta voltajes de hasta 25 voltios, aunque lo normal era trabajar entre -13V y 13V (excepto entre -3 y 3, que marcaba el cambio lógico de 0 a 1), es decir ni siquiera las polaridades coinciden. No obstante si se quisiera experimentar con ellos, existen adaptadores RS-232 para Arduino.

La mayoría de las placas llevan un conector USB y un convertidor que hace que podamos utilizar directamente el puerto USB del ordenador para comunicarnos con el puerto serie de la tarjeta. En caso de no incorporar este

convertidor, se necesitará un adaptador de TTL a USB. Normalmente las placas Arduino vienen con un solo puerto serie, también conocido como UART (*Universal Asynchronous Receiver/Transmitter*, transmisor/receptor universal asíncrono) o USART (*Universal Synchronous/Asynchronous Receiver/Transmitter*, transmisor/receptor universal síncrono/*asíncrono*), donde se encuentra el convertidor integrado y se corresponde a los pines 0 y 1 de las placas Arduino; esto quiere decir que en aquellas tarjetas en las que existan más de un puerto serie (actualmente sólo la gama Mega) sólo el puerto serie correspondiente a los pines 0 y 1 tendrá el convertidor integrado teniendo que utilizar adaptadores para usar los otros puertos.
Cuando se utilice la transmisión serie, los pines 0 y 1 estarán ocupados y no podrán ser utilizados como pines de entrada y salida digital.

Transmisión y recepción

Uno de los métodos que podemos utilizar para realizar para el envío y recepción de datos a la tarjeta Arduino es a través del puerto serie que viene integrado en ella.
Aunque en el mercado existen tarjetas con varios puertos serie, lo normal es que tengan sólo uno, que bien utilizado nos servirá casi siempre para cubrir nuestras necesidades; en caso de que fuera necesario, mediante código, se podrían habilitar otros pines para trabajar como puerto serie tal y como veremos en futuros ejemplos.
Ya hemos utilizado anteriormente el monitor serie tanto para enviar como para recibir datos de la tarjeta y lo hemos hecho sin necesidad de realizar complicados montajes ni cables a los pines 0 y 1; simplemente conectando un cable al puerto USB del ordenador.
Aunque no estemos utilizando ningún cable directamente sobre los pines 0 y 1, realmente los estamos utilizando "por debajo" y como veremos más adelante, no hace falta variar el *sketch* para trabajar con estos pines y con el USB ya que son lo mismo.
Todas las acciones referentes a la comunicación serie se realizarán con la clase `Serial`.
En caso de que exista más de un puerto serie (la placa Arduino Mega tiene 4) se pueden acceder mediante `Serial1`(pines 18 TX y 19 RX), `Serial2` (pines 16 TX y 17 RX) y `Serial3` (pines 15 TX y 14 RX). No obstante las tarjetas llevan la serigrafía RX y TX en los pines dedicados a la transmisión (TX) y recepción (RX) de las comunicaciones serie.

Antes de comenzar cualquier comunicación mediante el puerto serie, se debe configurar utilizando la instrucción `Serial.begin(velocidad);`, donde `velocidad` es la velocidad de transmisión que se utilizará durante la comunicación medida en bps. En cualquier momento podemos terminar la comunicación mediante `Serial.end()`.

Envío de mensaje serial

Anteriormente se han realizado ejercicios en el que se enviaban datos al monitor serie utilizando `Serial.print()` y `Serial.println()` (recordamos que la primera simplemente escribe el dato pasado como parámetro, mientras que la segunda añade una marca de nueva línea). Estas dos sentencias nos acompañaran durante gran parte de los desarrollos que hagamos con Arduino ya que son muy útiles para depurar los *sketch*. Debido a que no existe (por el momento) un depurador del programa al uso (uno donde podamos ejecutar paso a paso y ver el valor de las variables), nos valdremos de mensajes en el monitor para conocer el estado de las variables y el camino que sigue la ejecución del programa. Llenando la zona que queramos depurar de llamadas `Serial.println()` del tipo:

```
switch (var){
   case 0:
      llamadaFuncion1();
      Serial.println("1");
      if(var3){
         llamadaFuncion11();
         Serial.println("1.1");
      }
      else{
         llamadaFuncion12();
         Serial.println("1.2");
      }
   case 1:
      llamadaFuncion2();
      Serial.println("2");
      break;
   ....
   }
}
```

Si `var` vale 0 y `var3` 1 se ejecutarían las funciones `llamadaFuncion1`, `llamadaFuncion11` y `llamadaFuncion2`, y puede ser difícil de depurar, pero en la salida del monitor veríamos:

```
1
1.1
2
```

con lo que nos daríamos cuenta fácilmente de que nos falta un `break`. Más allá de ser utilizado para depurar, la utilidad en sí de la salida serie es transmitir datos desde la placa Arduino hacia el exterior, donde el receptor puede ser el monitor, otra placa...

Los datos que podemos transmitir son muy variados: números, caracteres, frases... pero realmente lo que se transmiten son 0 y 1. Arduino diferencia en dos tipos las transmisiones: las comprensibles por los humanos y las comprensibles por los dispositivos. Dentro de las transmisiones comprensibles por los humanos están las llamadas `Serial.print()` y `Serial.println()` donde lo que transmiten son códigos ASCII representación del dato en sí mientras que en las comprensibles por los dispositivos tenemos la función `Serial.write()`, que escribe los datos binarios en el puerto de salida a modo de bytes (uno o más). Para que veamos la diferencia podemos utilizar el sketch:

```
void setup(){
   Serial.begin(9600);
}
void loop (){
   Serial.print(64); // en el monitor aparece 64
   delay(1000);
   Serial.write(64); // en el monitor aparece @
   delay(1000);
}
```

Al realizar el `Serial.write()` aparece el carácter "@" porque estamos diciendo que lo que vamos a escribir es dato binario (no ASCII) es decir no le decimos que envíe el numero 64 tomado como "6" y "4", que es la representación de 64 para los humanos, sino un byte con el número 64 que es el valor ASCCI de la representación del carácter "@".

En caso de `Serial.print()` realmente se enviarían dos bytes, uno para el 6 (con valor 54) y otro para el 4 (con valor 52), es decir cada carácter se toma como byte independiente.

Las llamadas `Serial.print()` y `Serial.println()` permiten imprimir valores de cualquier tipo dentro del programa y además aceptan un segundo parámetro para modificar el formato de salida. Se puede modificar la base de representación de los enteros y los decimales en caso de punto flotante.

```
int a = 45;
float b = 2.3456789;
Serial.println(a);       // muestra 45
Serial.println(a, DEC);  // muestra 45
Serial.println(a, HEX);  // muestra 2D
Serial.println(a, OCT);  // muestra 55
Serial.println(a, BIN);  // muestra 101101
Serial.println(b);       // muestra 2.35
Serial.println(b,3);     // muestra 2.346
Serial.println(b,5);     // muestra 2.34568
```

En cuanto a la función `Serial.write()` la podemos utilizar con tres tipos de parámetros, un valor que se transmitirá como un byte, una cadena de texto que se transmitirá como una serie de bytes carácter a carácter o un array de bytes (también conocido como buffer de datos) y el tamaño del mismo que será transmitido también como una serie de bytes.

```
const int BUFF_SIZE = 256;
byte outByte[BUFF_SIZE];
...
outByte[4]=23;
outByte[5]=12;
outByte[6]=18;
...
Serial.write(outByte, BUFF_SIZE); // enviamos el array
Serial.write("fin de mensaje"); // envío de cadena de texto
Serial.write(10); // envío de bytes unitarios; nueva línea
Serial.write(13); // retorno carro
```

Hay que tener cuidado cuando se indica el tamaño del buffer a transmitir, ya que si el tamaño es mayor que el tamaño del buffer no da ningún error en compilación, simplemente cuando envíe, enviará los datos que estén en memoria más allá del fin del buffer, con lo que no sabemos que estaremos enviando y el funcionamiento será imprevisible.
Cuando se transmiten grandes cantidades de datos, es posible que tarde unos momentos en transmitirlos, pero el programa prosigue, es decir, el programa dice que se tiene que transmitir un array (por ejemplo) y se desentiende de la transmisión y sigue ejecutando instrucciones, es un proceso desatendido. Si se quiere esperar a que se terminen de transmitir los datos antes de seguir la ejecución del programa, se puede utilizar la llamada `Serial.flush()`, que como veremos más adelante tiene otro efecto en la lectura de datos.

Recepción de mensaje serie

La recepción del mensaje se realiza mediante un buffer (array de bytes) en la entrada del dispositivo que es capaz de almacenar hasta 64 bytes, esto significa que aunque no leamos el dato nada más llegar a la placa, éste no se pierde, sino que se almacena a la espera de ser leído; se comenzarán a perder datos en el momento que se superen los 64 bytes almacenados.
Antes de comenzar a leer, se debe saber si hay datos en la entrada esperando a ser leídos, para lo cual se usa la función `Serial.available()` que devuelve el número de bytes disponibles para su lectura. Una vez se sabe que existen datos en la entrada, la lectura se realiza mediante la llamada `Serial.read()` que devuelve el primer byte que esté disponible en el buffer de entrada; si no

hubiera datos devolvería un -1. Siempre que se realiza una lectura del buffer, los bytes leídos se eliminan del buffer; no se pueden volver a leer bytes ya leídos. Ejecutemos el siguiente *sketch* para ver qué es lo que nos devuelve la función `Serial.read()`:

```
byte inByte = 0;

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

void loop() {
   if (Serial.available() > 0) { // hay datos?
      inByte = Serial.read(); // lectura del buffer
      Serial.print("Se recibe: ");
      Serial.println(inByte);
   }
}
```

Si en el monitor escribimos 3 y se lo enviamos a la tarjeta, ésta responderá "Se recibe: 51". ¿Cómo es posible? Es porque realmente estamos leyendo el byte y no el número en sí, el byte contiene el número ASCII del 3, que es 51. En caso de que queramos obtener en la salida el carácter exacto introducido, debemos transformarlo, cambiando:

```
Serial.println(inByte);
```

por:

```
Serial.println(char(inByte));
```

Mediante la función `char()` obtenemos el carácter del código ASCII pasado como parámetro. Si lo que se transmite es un número entero, hay que tener aún más cuidado, porque imaginemos que se quiere transmitir desde el monitor el número "325"... es un entero, y un entero se guarda en memoria en 2 bytes, pero si se envían sus caracteres, lo que realmente se está enviando son 3 bytes, cada uno de ellos con el código ASCII de cada uno de sus dígitos por separado, entonces habría que leer el buffer byte a byte y multiplicar cada dígito por 10^n siendo n la posición que ocupa... o utilizar la función `Serial.parseInt()` que lo que hace es leer del buffer y devolver el primer entero que encuentre.
Variemos la función `loop()` para dejarla:

```
void loop() {
   if (Serial.available() > 0) { //hay datos?
      Serial.print("Se recibe: ");
      Serial.println(Serial.parseInt());
   }
}
```

Si ahora introducimos en el monitor "345", obtendremos como mensaje de salida el esperado "Se recibe: 345". Pero hemos dicho que devuelve el primer entero que encuentre, así si introducimos "1234eee12345" en el monitor... ¿qué obtendremos? por pantalla obtendremos:

```
"Se recibe: 1234"
"Se recibe: 12345"
```

Esto se debe a que el bucle sigue después de la primera llamada a `Serial.parseInt()` en la segunda llamada el buffer tendría como contenido "eee12345" (recuerde que los bytes leídos desaparecen) y la llamada a `Serial.parseInt()` devolvería el primer entero que encuentre, que sería "12345", perdiéndose el resto.
Del mismo modo tenemos `Serial.parseFloat()` para ayudarnos en la lectura de datos en punto flotante. Funciona del mismo modo que `Serial.parseInt()`, pero devolviendo el primer valor de punto flotante que encuentre y eliminando los bytes precedentes en caso de que no sean susceptibles de convertirse en parte del número.
Para lecturas de varios bytes a la vez tenemos estas dos funciones: `Serial.readBytes()` y `Serial.readBytesUntil()`. Mediante `Serial.readBytes()` podemos leer del buffer de entrada y escribir en un buffer propio un número de caracteres indicado. Por ejemplo para leer 20 caracteres:

```
char inData[20];
byte numBytes = Serial.readBytes(inData, 20);
```

Como podemos ver, el primer parámetro es el buffer (array de `char` o de `byte`) sobre el que se quiere escribir los bytes leídos y el segundo parámetro son los bytes a leer. En caso de que hubiera más bytes en el buffer de entrada, estos quedarían ahí esperando a una nueva lectura y en caso de que hubiera menos, la función se queda esperando leyendo hasta que se llegue al número de bytes indicados o se le acabe el tiempo de espera que se puede indicar con la función `Serial.setTimeout(milis)`, donde `milis` son los milisegundos que tiene que esperar como mucho a que entren nuevos datos; por defecto son 1000ms. `Serial.readBytes()` devuelve el número de bytes leídos; en el ejemplo anterior este dato lo guardaría en `numBytes`.
La variante `Serial.readBytesUntil()` incorpora un parámetro adicional (el primero de ellos) donde se le puede informar un carácter que cuando sea obtenido en la lectura deje de leer del buffer de entrada; si no aparece este carácter seguirá leyendo hasta el número de caracteres indicado en el parámetro o que se acabe el tiempo de espera, lo que ocurra antes.

```
char inData[20];
byte numBytes = Serial.readBytes('C',inData, 20); // termina de leer si
                                                  // encuentra 'C'
```

Para realizar búsquedas en el buffer de entrada disponemos de dos funciones la primera de ellas es `Serial.find()` que recorre el buffer de entrada hasta encontrar la cadena pasada como parámetro; ¡ojo! busca cadena y no carácter, por lo que hay que pasar el parámetro con comilla doble (o puntero a carácter para los aficionados a C). En:

```
Serial.find("op");
```

se buscará en el buffer hasta encontrar los bytes con valor 111 y 112 (ASCIIs de las letras 'o' y 'p') y que estén seguidos, los bytes leídos hasta encontrar la cadena son desechados. Esta función devuelve un booleano con el resultado de si lo ha encontrado o no. La segunda función para la búsqueda es muy parecida y se trata de `Serial.findUntil()` que tiene un parámetro más con un carácter que si lo encuentra durante la búsqueda en el buffer da la búsqueda como finalizada.

```
Serial.find("op", 'a'); //se detiene si encuentra una 'a'
```

Todas las funciones que se han visto de lectura del buffer de entrada, eliminan los bytes leídos, de modo que no se pueden nunca leer dos veces. En caso de que necesitemos tener una pre visualización del próximo byte para leer, se puede utilizar la función `Serial.peek()`, que es semejante a `Serial.read()` salvo que no elimina el byte del buffer, esto quiere decir que dos llamadas consecutivas a la función `Serial.peek()` siempre retornarán el mismo resultado.

```
byte bufferByte = Serial. peek();
```

Si no se quiere leer más datos del buffer, podemos eliminar todo lo que falte por leer mediante `Serial.flush()`, que dejará el buffer con 0 elementos pendientes. Si es necesario se pueden enviar y recibir caracteres no imprimibles (como el retorno de carro), para ello usaremos la notación de C, escapando algún carácter. El escape es mediante el carácter de contra barra '\' y aunque en la representación sean dos caracteres o más los que se escriben, realmente se envía un byte se envía uno... Por ejemplo:

Carácter a enviar	Significado	Valor del byte en hexadecimal
\'	Comilla simple	27
\"	Comilla doble	22
\?	Interrogación	3f
\\	Contra barra	5c
\0	Carácter nulo	00

Carácter a enviar	Significado	Valor del byte en hexadecimal
\a	Tono acústico	07
\b	Backspace	08
\f	Avance de página	0c
\n	Nueva línea	0a
\r	Retorno de carro	0d
\t	Tabulación horizontal	09
\v	Tabulación vertical	0b

Truco:

Las lecturas sobre el buffer de entrada borran los datos leídos de éste, si se necesita leer un byte sin extraerlo del buffer se debe usar Serial.peek().

Comunicación con el PC

La comunicación con el monitor es interesante desde el punto de vista de del desarrollo puesto que nos facilita muchas tareas, pero en la vida real el circuito que montemos no recibirá los datos de este monitor, sino que lo recibirá de teclados externos, otra tarjeta o por ejemplo de un ordenador. Aprovechando que disponemos de un puerto USB, podemos utilizarlo para realizar programas que se comuniquen con nuestra tarjeta.
Para crear la comunicación simplemente tenemos que seleccionar nuestro lenguaje de programación preferido y disponernos a crear un programa que sea capaz de leer o escribir en puertos serie. En el caso de algún lenguaje es posible que se deban instalar complementos y librerías para poder utilizar las comunicaciones serie.

Comunicación Arduino -> PC

Comenzaremos la comunicación entre Arduino y el PC mediante la lectura del PC de los datos que imprime Arduino en el monitor serie. Con esta lectura podemos hacer partícipe al ordenador de los datos disponibles en Arduino,

así por ejemplo se podrían medir temperaturas y mostrarlas en una pantalla de ordenador, leer los valores de unos potenciómetros y cambiar con ellos el volumen de audio, equipar a nuestra placa con un par de acelerómetros y controlar el ratón con ella...
Cuando se transmite entre dispositivos lo que se hace es codificar las instrucciones; por ejemplo supongamos que estamos en un caso en el que tenemos 3 acelerómetros que controlan volumen, agudos y graves, podemos enviar "v50a23g78" desde la tarjeta para saber en el ordenador que el volumen debe estar al 50%, los agudos al 23% y los graves al 78%; es decir estamos codificando la información y se debe descodificar siguiendo las mismas convenciones en el destino.
Además si todo el rato está enviando este tipo de mensajes la placa, aunque leamos un mensaje a medias sabemos cuándo empieza el siguiente mensaje: cuando recibamos una 'v'.
Para ver cómo recibir los datos desde el ordenador, haremos un ejercicio donde se realizará una lectura simple de los datos enviados y acto seguido se mostrarán por pantalla.
Lo que se haría en un programa "real" donde se utilicen los datos recibidos, sería interpretar lo enviado y hacer que el programa reaccione dependiendo de lo interpretado.
Como lenguaje de programación utilizaremos Java (aunque tal y como he comentado se podría utilizar cualquiera que pueda leer del puerto serie del ordenador).
Para utilizar el puerto serie en Java es necesario incluir unas librerías adicionales. Es posible utilizar las librerías *Javacomm* pero son más fáciles de conseguir las *RXTX* que son una implementación libre de la Java Communication API; de hecho las clases y las llamadas a las funciones son iguales que las de Javacomm y sólo cambia (externamente) el *package*. Estas librerías se pueden descargar desde `http://www.cloudhopper.com/opensource/rxtx/` para distintas plataformas (incluyen instrucciones de instalación).
El código Java de la aplicación que escucha sería:

```
import java.io.*;
import java.util.*;
import gnu.io.*; // si se usa Javacomm cambiar por import javax.comm.*

public class SerialRead implements Runnable, SerialPortEventListener {
   static CommPortIdentifier portId;
   static Enumeration<CommPortIdentifier> portList;

   InputStream inputStream;
   SerialPort serialPort;
   Thread readThread;
```

```
public static void main(String[] args) {
   portList = CommPortIdentifier.getPortIdentifiers();
   // se busca el puerto a abrir
   while (portList.hasMoreElements()) {
      portId =  portList.nextElement();
      if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
         if (portId.getName().equals("COM27")) { //"/dev/term/ttyS12"
            // se ha encontrado el puerto, se inicia la escucha
            SerialRead reader = new SerialRead();
            break;
         }
      }
   }
}

public SerialRead() {
   try {
      serialPort = (SerialPort) portId.open("SerialRead", 2000);
   } catch (PortInUseException e) {System.out.println(e);}
   try {
      inputStream = serialPort.getInputStream();
   } catch (IOException e) {System.out.println(e);}
   try {
      serialPort.addEventListener(this);
   } catch (TooManyListenersException e) {System.out.println(e);}
   serialPort.notifyOnDataAvailable(true);
   try {
      serialPort.setSerialPortParams(9600,
      SerialPort.DATABITS_8,
      SerialPort.STOPBITS_1,
      SerialPort.PARITY_NONE);
   } catch (UnsupportedCommOperationException e) {System.out.println(e);}
   readThread = new Thread(this);
   readThread.start();
}

public void run() {
   try {
      Thread.sleep(20000);
   } catch (InterruptedException e) {System.out.println(e);}
}

public void serialEvent(SerialPortEvent event) {
   switch(event.getEventType()) {
      case SerialPortEvent.BI:
      case SerialPortEvent.OE:
      case SerialPortEvent.FE:
      case SerialPortEvent.PE:
      case SerialPortEvent.CD:
      case SerialPortEvent.CTS:
      case SerialPortEvent.DSR:
      case SerialPortEvent.RI:
      case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
         break;
      case SerialPortEvent.DATA_AVAILABLE: // existen datos, los leemos
```

```
            byte[] readBuffer = new byte[20];
            try {
               while (inputStream.available() > 0) {
                  int numBytes = inputStream.read(readBuffer);
                  System.out.print(new String(readBuffer).trim());
               }
            } catch (IOException e) {System.out.println(e);}
            break;
      }
   }
}
```

Si finalmente se optó por las librerías *Javacomm*, hay que modificar el `import gnu.io.*` para que trabaje con `javax.comm.*`. A grandes rasgos, lo que hace el programa es primero buscar el puerto que se le ha indicado como puerto a utilizar, en este caso el "COM27", que sería el mismo que usamos en la configuración del entorno de Arduino. Si se utiliza algún Unix, se debe informar con la notación que le corresponde al puerto, por ejemplo "/dev/ttyS9". Dentro de la función `SerialRead()` se configura la conexión y mediante `serialPort.notifyOnDataAvailable(true)` decimos que cuando haya datos nos envíe un evento, que será atendido por la función `serialEvent()`, que en caso de ser un evento de tipo `SerialPortEvent.DATA_AVAILABLE` nos indicará que hay datos a la espera de ser leídos. La lectura se realiza mediante un buffer de 20 bytes de tamaño, por lo que si hay más de 20 bytes en la entrada el bucle se irá ejecutando hasta que no queden más, mientras se van mostrando por pantalla.

```
while (inputStream.available() > 0) {
   int numBytes = inputStream.read(readBuffer);
   System.out.print(new String(readBuffer).trim());
}
```

En un programa "útil" lo que se haría sería en este caso, sería preservar lo leído e ir procesándolo en lugar de mostrarlo por pantalla.
En cuanto a la parte Arduino como montaje físico no hay nada más que conectar la placa al ordenador (al puerto que hayamos informado en el programa) y cargar el siguiente *sketch*.

```
void setup() {
   Serial.begin(9600);
}
void loop() {
   Serial.print("Datos variados");
}
```

Si ejecutamos el programa Java tras haber cargado el *sketch*, veremos como en la salida por defecto de nuestro editor Java irá apareciendo el mensaje indicado en el *sketch*.

Comunicación PC -> Arduino

En la comunicación del PC con Arduino podremos enviar datos desde el PC hacia la placa como si fuera una entrada y así poder ejecutar unas instrucciones u otras. Podríamos tener por ejemplo un programa que controlara las persianas de casa y mediante una barra de desplazamiento en pantalla, pudiéramos subir y bajar las persianas dependiendo de la posición de la barra. La manera de trabajar con Java sería muy semejante a lo visto en el punto anterior, solo que en este caso se debe escribir en lugar de leer; se obtendría un objeto OutputStream del puerto serie y se utilizaría para escribir en él:

```
outputStream = serialPort.getOutputStream();
outputStream.write(64); // messageString.getBytes());
```

Como ya sabemos hacerlo en Java, en este punto haremos un ejercicio utilizando otro lenguaje de programación. Se había comentado en el primer capítulo que la programación de Arduino está basada en Processing, un lenguaje de programación diseñado para la enseñanza que ha ido evolucionando. Será precisamente con Processing con el que crearemos el programa que enviará datos a la tarjeta Arduino.
Para trabajar con Processing necesitamos su entorno de desarrollo, el cual está disponible en `http://processing.org/download`.

Nota:

Por el momento no se puede trabajar con comunicaciones serie en Processing si se está en un entorno de 64 bits.

Tras descomprimir el archivo, al ejecutar el programa veremos que es bastante semejante al entorno de programación Arduino (ya que es su descendencia). Lo que haremos es un programa con dos círculos que al pasar el ratón por encima de ellos enciendan distintos leds.
El programa en Procesing sería:

```
import processing.serial.*;

Serial myPort;  // objeto para trabajar con el puerto serie

int circleX, circleY;  // posición del círculo 1
int circle2X, circle2Y;  // posición del círculo 2
int circleSize = 50;   // diámetro
// colores a utilizar
color redColor;
color yellowColor;
color baseColor;
```

```
boolean circleOver = false;
boolean circle2Over = false;

void setup() {
   size(340, 260); // tamaño de pantalla
   // definición de los colores
   redColor = color(255,0,0);
   yellowColor = color(0,255,100);
   baseColor = color(102);
   // colocación de los círculos
   circleX = width/2-circleSize/2-10;
   circleY = height/2;
   circle2X = width/2+circleSize/2+10;
   circle2Y = height/2;
   ellipseMode(CENTER);
   println(Serial.list()); // obtiene la lista de puertos en pantalla
   // aquí se debe ajustar el puerto que corresponda en cada caso
   myPort = new Serial(this, Serial.list()[13], 9600);
}

void draw() {
   // Cuando se mueva el ratón se mirará si está dentro de cada circulo
   // si es así cambiar los colores y escribir carácter en el serial
   update(mouseX, mouseY);
   noStroke();
   if (circleOver) {
      background(redColor);
      myPort.write('A');
   } else if (circle2Over) {
      background(yellowColor);
      myPort.write('B');
   } else {
      background(baseColor);
      myPort.write('C');
   }

   // pintamos los círculos
   stroke(255);
   fill(redColor);
   ellipse(circleX, circleY, circleSize, circleSize);

   stroke(0);
   fill(yellowColor);
   ellipse(circle2X, circle2Y, circleSize, circleSize);
}

// guarda sobre qué círculo se está (si está sobre alguno)
void update(int x, int y) {
   if( overCircle(circleX, circleY, circleSize) ) {
      circle2Over = false;
      circleOver = true;
   } else  if( overCircle(circle2X, circle2Y, circleSize) ) {
      circleOver = false;
      circle2Over = true;
   } else {
      circleOver = circle2Over = false;
```

```
   }
}
// dadas unas coordenadas mira si están o no dentro del círculo
boolean overCircle(int x, int y, int diameter) {
   float disX = x - mouseX;
   float disY = y - mouseY;
   if(sqrt(sq(disX) + sq(disY)) < diameter/2 ) {
      return true;
   } else {
      return false;
   }
}
```

No entraremos a comentar el código ya que es muy sencillo y será fácil entenderlo; lo que sí que se ha de tener en cuenta es la configuración del puerto serie sobre el que escribiremos. Mediante la instrucción `println(Serial.list())` se listan en pantalla los puertos disponibles, debemos fijarnos en el número que ocupa el puerto en el que está conectada la placa y modificar la instrucción `myPort = new Serial(this, Serial.list()[13], 9600);` cambiando el 13 por el número que corresponda.

Tanto el *sketch* a utilizar como el montaje a realizar ya no deben tener secretos para nosotros. Se trata de un par de diodos puestos en las entradas 2 y 3 (ojo a la polarización de los diodos, usamos valores `LOW` para activarlos, por lo que el cátodo debe estar conectado a la entrada de la tarjeta). Para mayor realismo es mejor utilizar un led rojo y otro amarillo /verde y conectar el rojo a la salida 2 y el verde a la 3 de modo que concuerde con los colores que se mostrarán en la pantalla durante la ejecución del programa en Processing.

```
void setup(){
   Serial.begin(9600);
   pinMode(2,OUTPUT);
   pinMode(3,OUTPUT);
   digitalWrite(2,HIGH);
   digitalWrite(3,HIGH);
}

void loop (){
   if (Serial.available()>0){
      char c = Serial.read();
      if (c == 'A'){
         digitalWrite(2,LOW);
      }
      else if (c == 'B'){
         digitalWrite(3,LOW);
      }
      else{
         digitalWrite(2,HIGH);
         digitalWrite(3,HIGH);
      }
   }
}
```

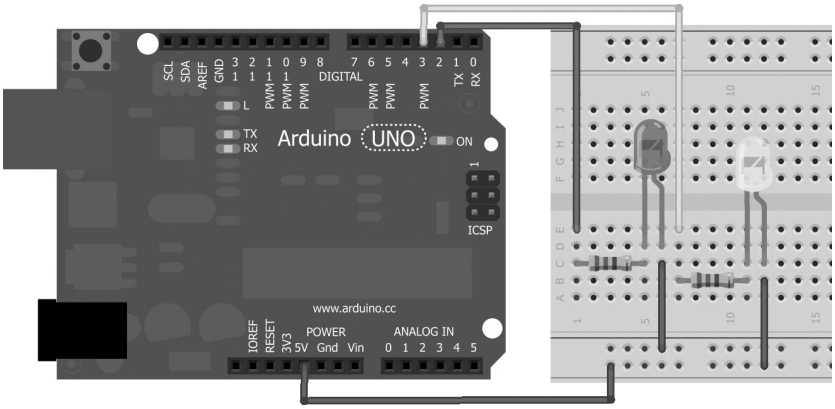


Figura 5.1. Circuito con dos leds para prueba de comunicación serie.

Se carga el *sketch* y con la tarjeta conectada al ordenador se ejecuta el programa en Processing (pulsando el botón de *play* o mediante el menú Sketck>Run), en ese momento aparecerá una pantalla como la de la figura 5.2 mostrando dos círculos, uno de color rojo y otro amarillo/verde; al mover el puntero del ratón sobre ellos se irán encendiendo entonces los leds correspondientes, apagándose al salir el puntero del círculo.

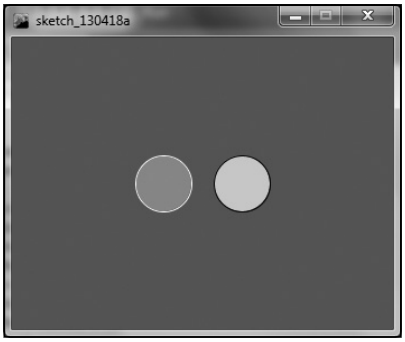


Figura 5.2. Pantalla creada mediante Processing.

Comunicación entre Arduinos

Además del puerto USB hemos dicho que existen un par de pines (concretamente el 0 y 1) que se pueden utilizar para realizar transmisión serie. Cada uno de los pines tiene una misión; el puerto 0 es el de recepción y está marcado con un RX mientras que el 1 es el de transmisión y está marcado

con un TX. La misión de estos pines es enviar o recibir los pulsos eléctricos que serán interpretados para tomar sentido dentro de Arduino. Los pulsos eléctricos se miden de 0 a 5V y para que todo sea correcto y no existan distorsiones en los datos por interferencias, lo ideal es que los dos sistemas que entren en la comunicación tengan el mismo nivel 0V, es decir que compartan el nodo tierra. En cuanto a lo que se refiere a la programación del *sketch*, no hay que hacer nada especial para que la información fluya hacia estos pines en lugar de hacia el USB como hacíamos hasta ahora, ya que de hecho están conectados. Si se utiliza una placa Mega y no se quieren utilizar los pines 0 y 1, entonces se debe cambiar la clase `Serial` por `Serial1`, `Serial2` o `Serial3` dependiendo de los pines que se quieran utilizar. Para comprender un poco mejor la transmisión usando los pines vamos a realizar unos ejercicios utilizando dos tarjetas Arduino. En caso de que dispongamos de una Mega se puede utilizar la misma tarjeta como emisora y receptora emulando dos tarjetas diferentes.

Comunicación unidireccional

Cuando hablamos de comunicación unidireccional, nos referimos a que uno de los actores transmite (y sólo transmite) y el otro de los actores recibe (y sólo recibe). En el caso de las tarjetas Arduino significaría que una de ellas sólo enviaría datos por el pin 1 y la otra simplemente recibiría datos por el pin 0. Así pues esta es la configuración más sencilla en cuanto al montaje de las dos tarjetas, un cable que una el pin 1 TX del transmisor con el pin 0 RX del receptor.

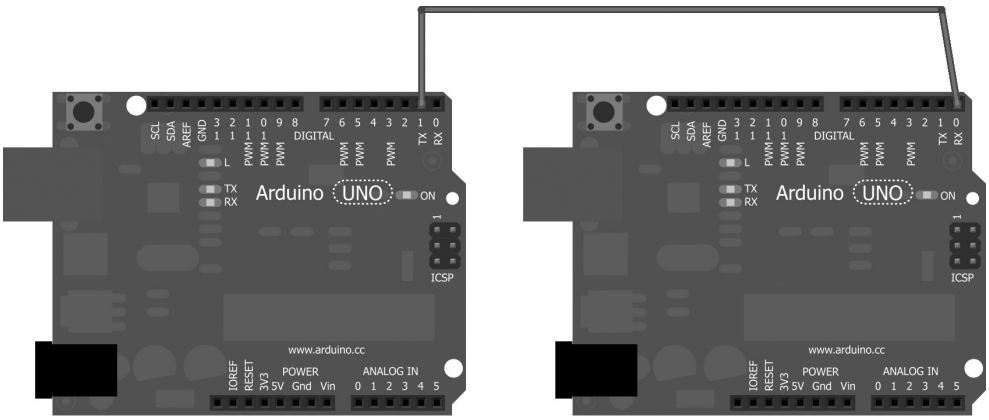


Figura 5.3. Conexión para transmisión con un cable.

En este ejemplo necesitaremos dos *sketch*, uno para el emisor y otro para el receptor. El *sketch* del emisor podía ser:

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

void loop (){
   // transmitimos para que un segundo esté el led
   // encendido y un segundo apagado
   Serial.print('H');
   delay(1000);
   Serial.print('L');
   delay(1000);
}
```

Y el del receptor se encargará de hacer que el led 13 se encienda cada vez que reciba un carácter 'H'.

```
void setup(){
   Serial.begin(9600);
   pinMode(13,OUTPUT);
}

void loop (){
   if (Serial.available() > 0) {
      // Leemos la entrada
      byte incomingByte = Serial.read();
      // Mostrar lo recibido:
      Serial.print("He recibido: ");
      Serial.println(incomingByte, DEC);
      // si es H encender el led, si no apagarlo
      if (incomingByte == 'H'){
         digitalWrite(13,HIGH);
      }
      else{
         digitalWrite(13,LOW);
      }
   }
}
```

Para probarlo, cargamos el *sketch* del emisor en la tarjeta correspondiente y la del receptor en el suyo y luego podemos comprobar el monitor de cualquiera de ellas donde se producirán salidas (además del parpadeo del led del receptor). Recomiendo cargar los programas antes de conectar los cables entre las placas, ya que si se hace la carga con los cables conectados pueden producirse problemas de comunicación con el ordenador y fallar la carga del programa.
En el monitor serie del emisor veremos *HLHLHL*... mientras que en el receptor veremos *He recibido: 72* y *He recibido: 76* dependiendo del carácter recibido a la vez que se enciende y apaga el led correspondiente al pin 13.

Anteriormente se ha explicado que para una mejor transmisión, el emisor y el receptor deben tener el mismo nivel 0V o dicho de otro modo las masas deben estar unidas. El hecho de que las placas se encuentren juntas hace que el nivel de ruido al que están sometidas sea semejante pero no está de más añadir un nuevo cable entre cualquiera de los pines tierra de la placa emisora (pin GND) y cualquiera de los pines tierra de la placa receptora, así mejoraremos la comunicación.

Comunicación bidireccional

La comunicación bidireccional se da cuando los dos actores pueden tanto enviar como recibir datos. En nuestro caso, dado que ambas tarjetas podrán tanto enviar como recibir deben tener los pines 0 y 1 conectados, y los deben tener de manera que el pin de emisión de una tarjeta esté conectado con el pin de recepción de la otra.

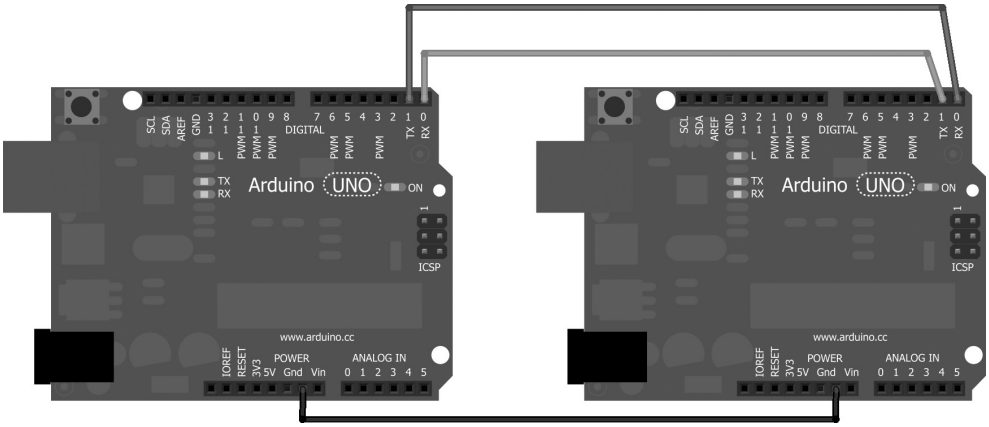


Figura 5.4. Conexión para transmisión con tres cables.

Los *sketches* que utilizaremos en este ejemplo deben tanto emitir como recibir. Uno de ellos se encargará de pensar un número y el otro se encargará de adivinarlo.

En el *sketch* de la tarjeta que comienza la partida (la que piensa el número a adivinar) debemos crear un número al azar y enviar a la tarjeta jugadora que se comienza la partida, esto lo haremos con el carácter 'G', es decir una vez que se envía la letra 'G' la segunda tarjeta comenzará a enviar números y la primera tarjeta debe decir si son mayores o menores hasta adivinar el número, momento en el que se encendería el led 13.

El *sketch* de la tarjeta que va a pensar el número sería:

```
int number;

void setup(){
   Serial.begin(9600);
   pinMode(13,OUTPUT);
   digitalWrite(13,LOW);
   randomSeed(analogRead(A0)); // inicializa el random
   number = random(1000); // número a adivinar
   // podríamos imprimir el número pensado
   // Serial.print(number);
   Serial.write('G'); // comienza la partida
}

void loop (){
   if (Serial.available() > 0) {
      int readInt = Serial.parseInt(); // leer el número indicado
      if (readInt>number){
         Serial.write('H');
      }
      else if (readInt<number){
         Serial.write('L');
      }
      else{
         digitalWrite(13,HIGH); //acertado
      }
   }
}
```

Como podemos ver en el código, simplemente inicia la partida con la letra 'G' una vez que tiene el número pensado, lee la entrada de datos y va mirando si es mayor o menor y le indica 'H' si el número pensado es mayor o 'L' si es menor.

El *sketch* que adivinará el número por su parte comenzará con un número al azar entre 0 y 1000 (límites que tiene también el primer *sketch*, y según vaya respondiéndole si es mayor o menor ajustará dos variables `maxVal` y `minVal` para conseguir el valor pensado por la otra tarjeta.

```
int numberSent;
int maxVal;
int minVal;

void setup(){
   Serial.begin(9600);
   pinMode(13,OUTPUT);
   digitalWrite(13,LOW);
   randomSeed(analogRead(A0)); // inicializa el random
}

void loop (){
   if (Serial.available() > 0) {
```

```
      // buscamos la G
      byte readChar = Serial.read();
      if (readChar == 'G'){ // nuevo juego iniciamos las variables
         minVal = 0;
         maxVal = 1000;
      }
      else{
         if (readChar == 'L'){
            maxVal = numberSent;
         }
         else
         if (readChar == 'H'){
            minVal = numberSent+1; // minVal es inclusivo en el random()
         }
         else{
            digitalWrite(13,HIGH); // Lectura no entendida!
         }
      }
      numberSent = random(minVal, maxVal);   // número aleatorio entre min
                                             // y máximo
      Serial.println(numberSent); // enviamos el número pensado
      delay(1000);
   }
}
```

En caso de que se lea algún byte que no sea ni 'G' ni 'H' ni 'L' es que ha leído algo que no sabe interpretar (posiblemente valores captados por ruido); si es así se mostrará encendiendo el led 13.

Lo mejor de este ejercicio es que si no se tienen dos placas, al menos podemos jugar a adivinar números en cualquiera de los dos bandos, dependiendo del *sketch* que carguemos podemos adivinar o bien que la tarjeta nos adivine el número desde el monitor serie, simplemente hay que acordarse de comenzar la partida con 'G'.

6 Salida de audio

En este capítulo aprenderá a:

- Conocer los tipos de altavoz más usados.
- Usar un altavoz piezoeléctrico.
- Emitir sonidos en Arduino.
- Componer melodías.
- Controlar los tonos de los sonidos.
- Reproducir sonidos MIDI.

Aunque Arduino no está pensado para la creación de sonidos digitales ya que no posee ningún sintetizador integrado, si podemos aprovechar su versatilidad para llegar a producir melodías tanto en altavoces como en dispositivos MIDI (*Musical Instrument Digital Interface*, Interfaz Digital de Instrumentos Musicales).
Como ya será conocido, Arduino se comunica con el mundo físico a través pulsos eléctricos, pero los pulsos eléctricos no son audibles a no ser que utilicemos algún elemento que transforme esos pulsos eléctricos en sonido audible. Si somos capaces de oír un sonido es porque algo ha generado una onda sonora (normalmente una presión de aire o golpe de viento) que se propaga y llega a nuestros oídos quien transforma la onda de modo que sea interpretada por el cerebro. Nos valdremos entonces de herramientas para generar esta onda sonora a partir de un pulso eléctrico. Las herramientas de las que nos valdremos en los ejemplos para conseguir sonidos serán en este caso los altavoces piezoeléctricos y dispositivos MIDI.

Salida mediante altavoz

Una de las maneras más sencillas de transformar los pulsos eléctricos en ondas sonoras es mediante altavoces.
La manera de funcionar de un altavoz es la siguiente: se recibe un pulso eléctrico en sus terminales, este pulso genera una reacción normalmente mecánica, que dependiendo del tipo de altavoz será distinta, y por último esta reacción mecánica genera presiones en el aire que rodea el elemento físico generador de la reacción mecánica y estas presiones de aire son las que generan las ondas de sonido, que pueden ser audibles o no.
El que un sonido sea audible o no para los humanos, lo fija la frecuencia de éste. Hemos comentado que los sonidos son ondas y las ondas tiene una frecuencia que es el número de veces que se producen dichas ondas en una unidad de tiempo; la frecuencia se mide en Hercios (en honor a Heinrich Rudolf Hertz). Los Hercios (Hz) son el número de veces que se repite algo en un segundo y en este caso lo que se repite es la onda sonora. La función inversa de la frecuencia es el periodo que se refiere a cuánto tiempo tarda en repetirse la onda.
Para que un sonido sea audible por el oído humano debe estar entre los 20Hz y los 20kHz (20000Hz) o bien entre los 6Hz y los 16kHz, dependiendo de las fuentes y es que el rango de sonidos que puede escuchar el oído humano varía mucho dependiendo de la persona e incluso de las edades, donde a medida que avanza y se es más mayor se va perdiendo rango de frecuencia audible.

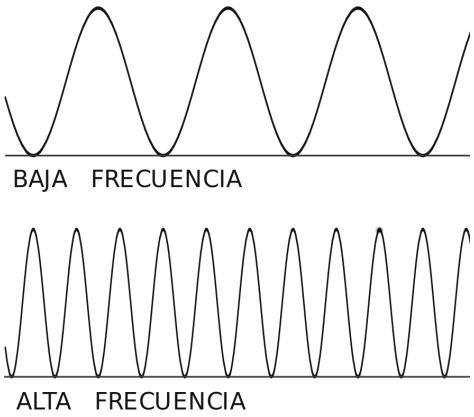


Figura 6.1. Frecuencias de onda.

Los altavoces más utilizados son:

- Altavoz dinámico: Formado por una bobina o solenoide central que en su interior tiene alojado un imán, al pasar la corriente eléctrica por el solenoide se crean unos campos magnéticos en su interior que atraen o repelen la pieza magnética haciendo que ésta se mueva a lo largo de su interior. La pieza magnética va apoyada o adherida a una membrana normalmente de plástico que al ser desplazada por la pieza magnética crea el golpe de aire necesario para generar el sonido. Son ideales para un rango de frecuencias muy amplio en especial en el rango audible, no obstante cambiando el material de la membrana que hace de diafragma se obtienen diferentes resultados. Este tipo de altavoces podemos encontrarlos en el PC o en muchos de los auriculares.
- Altavoz piezoeléctrico: La piezoelectricidad es una propiedad de algunos materiales de verse deformados cuando se les aplica una diferencia de potencial. Solamente ocurre con ciertos materiales cerámicos y cristalinos que además presentan el efecto inverso, que cuando se les aplican fuerzas mecánicas aparecen campos eléctricos en ellos, pero esta característica no nos importa en este momento. La deformación que tienen estos materiales está perfectamente reglada por unas fórmulas que no veremos pero que indican que la deformación causada es proporcional a la tensión mecánica ejercida en ellos y al campo eléctrico aplicado, y de modo inverso el desplazamiento eléctrico es proporcional al campo eléctrico aplicado y a la tensión mecánica ejercida. Para generar sonido una de sus caras se une a una cono de ampliación que será desplazado generando presión en el aire que generará las ondas sonoras emitiendo pequeños "clicks"

como si fuera un tambor. Si se cambia la frecuencia de los pulsos eléctricos enviados variará el número de "clicks" 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.

Figura 6.2. Altavoz dinámico y piezoeléctrico.

Para comenzar realizando un poco de ruido haremos un primer ejercicio en el que simplemente emitiremos un sonido, nada espectacular para ello utilizaremos un altavoz piezoeléctrico.
Cuando se monte el altavoz piezoeléctrico hay que fijarse en la polaridad de éste, que normalmente viene marcada por el color de los cables si es un altavoz piezoeléctrico de disco o por alguna serigrafía en el componente. Al montaje se le puede añadir un led en serie con el altavoz de modo que cuando suene el altavoz se ilumine el led, así se tendrá señal visual además de acústica. Si el sonido del altavoz es muy fuerte (molesta al resto de las personas de la casa) se puede poner en serie con él una resistencia; cuanto mayor sea la resistencia menor será el volumen al que suene. En caso de no tener un altavoz piezoeléctrico pasivo (dos terminales) si se tiene uno activo (tres terminales) lo único que se debería hacer de diferente es polarizar correctamente el altavoz y podría ser utilizado igualmente en todos los ejemplos del capítulo.
Aunque en Arduino existe una función para la generación de tonos llamada `tone()`, nosotros en este ejercicio los realizaremos a mano.
Para realizar diferentes tonos se envía al pin de salida un pulso eléctrico con diferente frecuencia y dependiendo de la frecuencia aplicada se obtendrá una "nota musical" u otra diferente. Para calcular el tiempo que debe estar la salida en `HIGH`, se debe pensar que un periodo es el tiempo que tarda en repetirse la onda, luego si la onda es `HIGH` y `LOW` a partes iguales, el tiempo en `HIGH` será la mitad del periodo y sabiendo que la frecuencia es la inversa del periodo tendríamos:

```
tiempoEnHigh = periodo / 2 = 1 / (2 * frecuencia)
```

Así la nota La, tomada como nota de referencia, son 440 Hz, luego tendríamos un periodo de 2272ms y un tiempo de 1136 ms en `HIGH`.

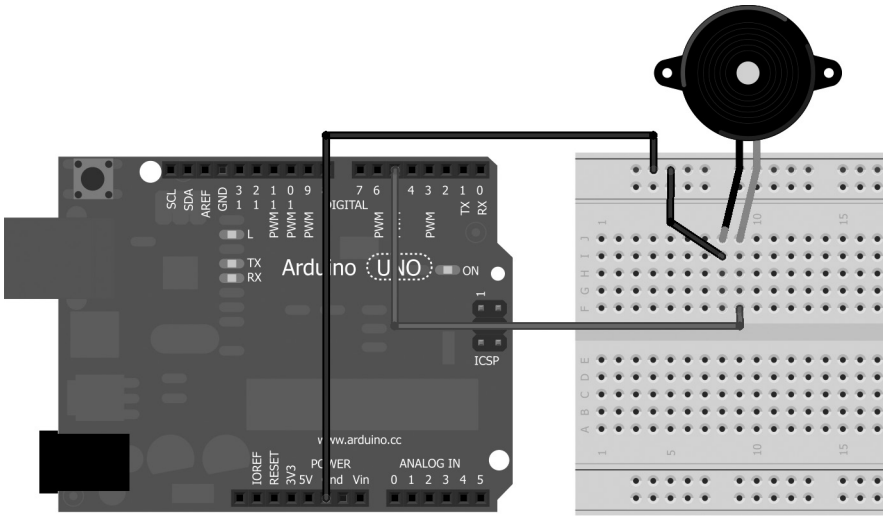


Figura 6.3. Montaje con altavoz piezoeléctrico.

Nota musical	Nota	Frecuencia	Periodo	Tiempo en HIGH
Do	c	261 Hz	3830	1915
Re	d	294 Hz	3400	1700
Mi	e	329 Hz	3038	1519
Fa	f	349 Hz	2864	1432
Sol	g	392 Hz	2550	1275
La	a	440 Hz	2272	1136
Si	b	493 Hz	2028	1014
Do	C	523 Hz	1912	956
Re	D	587 Hz	1703	851
...				

Haremos que suene la nota "La" cada segundo. Para ello generamos el *sketch* siguiente:

```
int speakerPin = 5;
void setup() {
   pinMode(speakerPin, OUTPUT);
}
void loop() {
   playTone(1136, 1000);
```

```
   delay(1000);
}

void playTone(int toneToPlay, int duration) {
   for (long i = 0; i < duration * 1000L; i += toneToPlay * 2) {
      digitalWrite(speakerPin, HIGH);
      delayMicroseconds(toneToPlay);
      digitalWrite(speakerPin, LOW);
      delayMicroseconds(toneToPlay);
   }
}
```

En el `loop` principal simplemente llamamos a la función `playTone()` con dos parámetros: el tiempo a estar en `HIGH` (que es la nota a sonar) y la duración de la misma. Dentro de la función `playTone()`, lo que se hace es mantener la salida a `HIGH` durante el tiempo marcado por la frecuencia, como se quería tener una frecuencia de 440Hz esto implica según la fórmula un periodo de 2272ms o lo que es lo mismo se debe tener la mitad del tiempo en `HIGH` y la mitad en `LOW`, por eso se pasa a la función el valor 1136, que serán los milisegundos de señal. El bucle `for` se encarga de controlar la duración de la nota.

Hay que fijarse en que en la condición se ha puesto `duration * 1000L`, esto es porque `duration` es un `int` y 1000 como constante sería también 1000, lo que da una alta posibilidad de que la multiplicación desborde al entero; al colocar la `L` nos aseguramos una operación cuyo resultado sea un `long`. El incremento del `for` se realiza sumando dos veces a `i` el tiempo de estar en `HIGH` ya que en el interior del bucle existen dos `delays` uno para `HIGH` y otro para `LOW` del mismo tiempo, así que para contabilizar el tiempo hay que tener en cuenta los dos.

Como podemos imaginar, la función `delayMicroseconds()` es semejante a `delay()` salvo que el parámetro en lugar de representar milisegundos a esperar, son microsegundos. Ya podemos probar nuestro primer montaje creador de ruidos. Invito a que se hagan pruebas variando el primer parámetro de la llamada `playTone()` con diferentes valores (no tienen por qué ser los de la tabla) para comprobar los diferentes tonos conseguidos.

Reproductor de melodías

Si se espera que en este punto hagamos un reproductor capaz de hacernos tirar nuestro reproductor de música estamos equivocados, más bien lo que conseguiremos en este punto es realizar un reproductor semejante a los que vienen en las tarjetas que cuando se abren suenan... si correcto, esas tan molestas.

Comenzaremos reproduciendo la escala musical, para ello nos valdremos del mismo montaje que en el ejercicio anterior, pero en el *sketch* haremos varias modificaciones.
Necesitaremos una serie de variables globales y constantes entre ellas:

```
const byte names[] = {'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};
```

Donde guardaremos el nombre de las notas de modo que nuestra canción se compondrá utilizando estas letras. La tabla de conversión con las notas reales será la de la tabla anterior.
En este caso tenemos definidas de Do a Sol y la 'C' corresponde a Do completando la escala.

```
const  int tones[] = { 1915, 1700, 1519, 1432, 1275, 1136, 1014, 956 };
```

Son los tonos de cada una de las notas anteriormente definidas, se convertirá en la duración que debe tener el pulso enviado al altavoz.

```
const int beats[] = {1,1,1,1,1,1,4,3,1,1,1,1,1,1,4 };
```

Mediante este array se guardan las duraciones de cada una de las notas a tocar. No tiene que ver con la frecuencia sino con el tiempo que se mantiene cada nota reproduciéndose en el altavoz y está ligado al tempo de reproducción.

```
static byte notes[] = "cdefgab bagfedc"; // notas a tocar
```

Este array nos servirá de partitura, cada letra representa a una nota musical definidas anteriormente por su tono, mientras que el espacio servirá para añadir una pausa. La manera en la que haremos música será tomando cada una de las notas aquí representadas y viendo cuánto tiempo se debe tener en `HIGH` para generar el sonido a través del array `tones`, por último el tiempo que se debe estar reproduciendo viene dado por `beats` y todo ello junto genera el sonido deseado y nota tras nota nuestra melodía, en este caso la escala creciente y luego decreciente.
También crearemos otras variables para contener el pin utilizado para la salida, el número de notas a reproducir y el tempo de reproducción (término musical referido a la velocidad de reproducción de las notas).
En el `setup()` simplemente configuramos el pin `speakerOut` como de salida. El `loop()` principal recorrerá las notas a reproducir obteniendo su duración y las irá reproduciendo una a una con una pausa entre ellas dada por el tempo de la reproducción.

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

void loop() {
```

```
   for (int i = 0; i < length; i++) {
      playNote(notes[i], beats[i] * tempo);
      // pausa entre notas
      delay(tempo / 2);
   }
}
```

Como puede verse necesitaremos una función llamada `playNote()`. Esta función se encargará de traducir la nota leída a su tono para así aprovechar la función hecha en el ejercicio anterior (la función `playTone()`), además controlará si el carácter leído es un espacio en blanco y en tal caso hará una pausa de duración dada por su valor correspondiente en `beats`, como si fuera una nota más pero sin reproducirla.

```
void playNote(byte note, int duration) {
   if (note == ' '){
      delay(duration); // no tocar nota
   }
   else{
   // reproducir la nota correspondiente
      for (int i = 0; i < 8; i++) {
         if (names[i] == note) {
            playTone(tones[i], duration);
         }
      }
   }
}
```

El código completo sería:

```
int speakerPin = 5;
int length = 15; // numero de notas
static byte notes[] = "cdefgab bagfedc"; // notas a tocar
// duración de las notas
const int beats[] = {1,1,1,1,1,1,4,3,1,1,1,1,1,1,4 };
// nombre de las notas
const byte names[] = {'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};
// duración de cada tono
const  int tones[] = { 1915, 1700, 1519, 1432, 1275, 1136, 1014, 956   };
// velocidad de reproducción
int tempo = 120;

void setup() {
   pinMode(speakerPin, OUTPUT);
}

void loop() {
   for (int i = 0; i < length; i++) {
      playNote(notes[i], beats[i] * tempo);
      // pausa entre notas
      delay(tempo / 2);
   }
}
```

```
/*****************************************
* Reproduce durante un tiempo dado por el parámetro
* duration el tono dado por el parámetro toneToPlay
*****************************************/
void playTone(int toneToPlay, int duration) {
   for (long i = 0; i < duration * 1000L; i += toneToPlay * 2) {
      digitalWrite(speakerPin, HIGH);
      delayMicroseconds(toneToPlay);
      digitalWrite(speakerPin, LOW);
      delayMicroseconds(toneToPlay);
   }
}

/*****************************************
* Obtiene el tono para la nota a tocar
*****************************************/
void playNote(byte note, int duration) {
   if (note == ' '){
      delay(duration); // no tocar nota
   }
   else{
   // reproducir la nota correspondiente
      for (int i = 0; i < 8; i++) {
         if (names[i] == note) {
            playTone(tones[i], duration);
         }
      }
   }
}
```

Si cargamos el *sketch* podremos oír la escala musical creciente, una pausa, la escala musical decreciente y vuelta a empezar. Podemos jugar variando los tonos para ver como se comportarían los sonidos (por ejemplo cambiar de octava) o modificar el tempo para variar la velocidad de reproducción.

Reproductor de melodías completo

Creo que estaremos todos de acuerdo en que el sonido producido por el ejercicio anterior dista mucho de ser música. En el siguiente ejercicio reproduciremos una pieza de Johannes Brahms o bueno... más bien se intentará simular una pieza clásica de él, en concreto una pieza realizada en honor al nacimiento del hijo de una amiga que es conocida por ser una de las canciones de cuna más famosas. Debo decir que mis conocimientos de música son nulos y la composición está hecha muy de oído, así que pido disculpas por adelantado, pero cumplió su objetivo que era calmar a mi hija.
Como puede que nos interese reducir el volumen para que el bebé se vaya durmiendo incorporaremos un potenciómetro con esta funcionalidad y además un botón para encender y apagar la reproducción, así cuando el

bebé esté dormido podemos dejar de reproducir (sí también se puede desenchufar, pero queda menos profesional). En el montaje del circuito usaremos un led, una resistencia de 220Ω, un pulsador, un altavoz piezoeléctrico y un potenciómetro. El valor del potenciómetro en un principio nos da igual, pero se ha de tener en cuenta que cuando el valor de la resistencia esté a 0 será el máximo volumen, mientras que a medida que se vaya incrementando la resistencia se irá disminuyendo el volumen, lo que implica que si se usa un potenciómetro con valor nominal muy alto, pronto nos quedaremos sin volumen; uno de 5kΩ está bien.

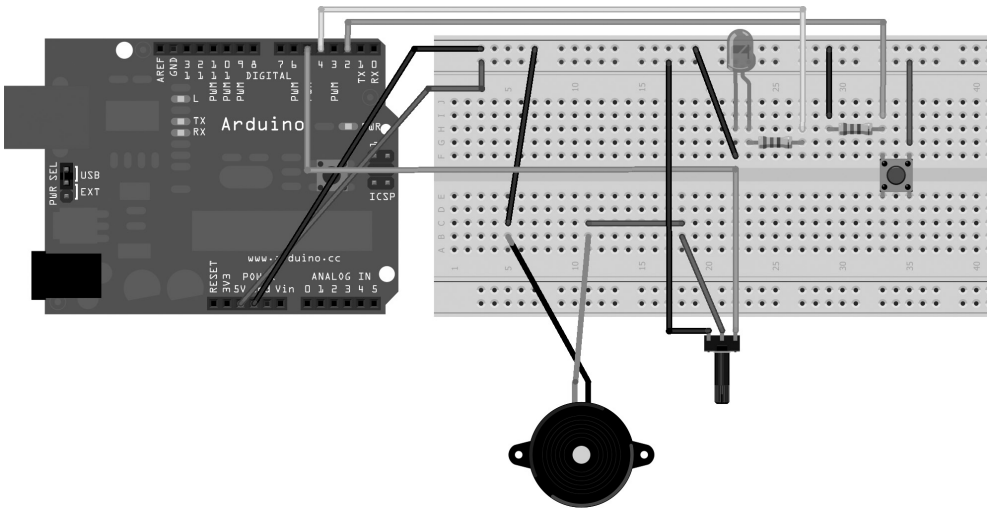


Figura 6.4. Circuito para reproducción con volumen y parada.

Si prestamos atención al circuito final, podemos observar tres circuitos más pequeños que conforman el montaje, todos ellos vistos anteriormente:

- El primer circuito sería referente a la reproducción, que seguiremos haciéndola mediante el pin 5, pero en este caso introduciremos un potenciómetro entre la salida y el altavoz piezoeléctrico para controlar el volumen, este circuito se debe realizar sobre la pata central y una de las laterales, llevando la otra pata hacia masa.
- Por otro lado se debe montar un circuito de salida para el led, en este caso hemos optado poner una resistencia externa en lugar de utilizar la interna como en otros montajes y se ve envuelto el pin 4.
- Por último existe un circuito para la detección de la pulsación del botón en el que entra en juego el pin 2.

Los tres circuitos que anteriormente habíamos visto por separado en este caso funcionan al unísono completándose.
El *sketch* será muy semejante al ejemplo anterior pero habrá que modificar ciertos aspectos como incluir el control del led o el control de la pulsación del botón y cómo no, el cambio de la partitura. En cuanto a la pulsación del botón lo más rápido sería introducir un `digitalRead ()` en el `loop()`, concretamente debería ser dentro del `for`, ya que si lo pusiéramos fuera habría que esperar toda la canción antes de que se apagara; en lugar de esto crearemos una función aparte para mejorar la lectura y estructura del código.
Lo primero sería añadir algunas variables que nos ayudarán a saber cuál es la última nota reproducida, así cuando paremos y volvamos a poner en marcha mediante el botón, continuará en la última nota donde lo dejó, también algunas variables para controlar si se debe escuchar la lectura del botón o no (para evitar que si se deja el botón apretado se encienda y apague la música constantemente).

```
int speakerPin = 5;
int ledOut = 4;
int length = 32; // numero de notas
int currentNote =0; // última nota tocada
boolean runPrograms = false;
boolean canListenButton = true;
boolean button = 2; // pin del botón
```

Los arrays con las frecuencias y nombres de las notas serán los mismos, pero en cuanto a la melodía y los tiempos de cada nota, los nuevo valores serán:

```
byte notes[] = "cce cce ceCgffe ccd ccd cdgfegC ";
int beats[] = {2,2,6,  1,  2,2,6,  1,  1,2,5,3,3,3,6,  1,  2,2,6,  1,
2,2,6,  1,  2,2,2,2,3,4,6,  4 };
```

Los espacios introducidos en `beats` son intencionados para permitir mejor lectura de los tiempos en relación a las notas, pero a la hora de reproducir no tienen efecto, los que sí tienen efecto son los espacios en `notes`.
En el `setup()` se deberían preparar los pines de salida para el led y el altavoz y de entrada para el pulsador y el `loop()` se encargaría como siempre del grueso del programa.

```
void loop() {
   int state = digitalRead(button); // lectura del botón
   // si está escuchando y es 1 parar o encender la reproducción
   if (state == HIGH && canListenButton) {
      canListenButton = false;
      runPrograms = !runPrograms;
   }
   else{
      if (state == LOW){
         canListenButton = true;
```

```
      }
   }
   ...
```

En esta función debemos controlar el momento en el que se pulsa el botón para comenzar a reproducir o dejar de hacerlo y evitar a la vez que si se mantiene el botón apretado comience a reproducir y se pare y así sistemáticamente. Lo que primero que se hace es comprobar el estado del pulsador, si el estado es `HIGH` y se debe escuchar al botón, entonces acto seguido hay que dejar de escuchar al botón (mediante la variable `canListenButton`) y comenzar a reproducir si no se está reproduciendo y parar la reproducción si se está reproduciendo, esto se controla con las variable `runPrograms`. Si el estado de pulsación es `HIGH` y no se debe escuchar al botón, no se debe hacer nada. Si el estado de la pulsación es `LOW` quiere decir que no está apretando el botón o que lo ha liberado, así pues pasamos a volver a escuchar al botón. Una vez tenemos configurada la variable `runPrograms`, podemos saber qué hacer con las salidas.

```
// si se debe reproducir la música
if (runPrograms){
   executePrograms();
}
else{
   // no se debe reproducir, apagar el led
   digitalWrite(ledOut, LOW); // led apagado
}
```

En caso de que sea `true` se debe reproducir la música y encender el led mientras que si es false se debe apagar el led.
La función `executePrograms()` se encarga de realizar todas las tareas que impliquen que el reproductor esté en marcha, en nuestro caso encender el led y reproducir la nota, pero si se hubiera implementado el volumen por software, aquí sería también un buen lugar desde controlar dicho nivel de volumen de salida.

```
void executePrograms() {
   digitalWrite(ledOut, HIGH); // led encendido
   playMusic();
}

void playMusic(){
   playNote(notes[currentNote], beats[currentNote] * tempo);
   // pausa entre notas
   delay(tempo / 2);
   currentNote ++;
   if (currentNote == length){
      currentNote = 0;
   }
}
```

Mediante la función `playMusic()` se obtiene la nota que actualmente se debe reproducir y su duración y se reproduce mediante la función `playNote()` ya usada anteriormente; acto seguido se aumenta el índice `currentNote` para apuntar a la próxima nota a reproducir, siempre controlando no reproducir notas fuera del buffer, en tal caso la nota a reproducir volverá a ser la 0. En cuanto a las funciones `playNote()` y `playTone()` quedarían iguales que el ejemplo anterior. El código completo tendría el aspecto:

```
int speakerPin = 5;
int ledOut = 4;
int length = 32; // numero de notas
int currentNote =0; // última nota tocada
boolean runPrograms = false;
boolean canListenButton = true;
boolean button = 2; // pin del botón
// notas a reproducir
byte notes[] = "cce cce ceCgffe ccd ccd cdgfegC ";
// duración de las notas
int beats[] = {2,2,6,  1,  2,2,6,  1,  1,2,5,3,3,3,6,  1,  2,2,6,  1,
2,2,6,  1,  2,2,2,2,3,4,6,  4 };
// nombre de las notas
const byte names[] = {'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};
// duración de cada tono
const  int tones[] = {1915, 1700, 1519, 1432, 1275, 1136, 1014, 956   };
int tempo = 130; // velocidad de reproducción

void setup() {
   pinMode(speakerPin, OUTPUT); // altavoz
   pinMode(button, INPUT); // botón
   pinMode(ledOut, OUTPUT); // led
   digitalWrite(ledOut, LOW);
}
void loop() {
   int state = digitalRead(button); // lectura del botón
   // si está escuchando y es 1 parar o encender la reproducción
   if (state == HIGH && canListenButton) {
      canListenButton = false;
      runPrograms = !runPrograms;
   }
   else{
      if (state == LOW){
         canListenButton = true;
      }
   }
   if (runPrograms){
      // si se debe reproducir la música
      executePrograms();
   }
   else{
      // no se debe reproducir, apagar el led
      digitalWrite(ledOut, LOW); // led apagado
   }
}
```

```
/*******************************************************
* Si reproduce la música, encender el led
*******************************************************/
void executePrograms() {
   digitalWrite(ledOut, HIGH); // led encendido
   playMusic();
}

/*******************************************************
* Reproduce la música, nota a nota, guardando la
* nota actual para conecer en la posición de la melidía
* en la que se encuentra
*******************************************************/
void playMusic(){
   playNote(notes[currentNote], beats[currentNote] * tempo);
   // pausa entre notas
   delay(tempo / 2);
   currentNote ++;
   if (currentNote == length){
      currentNote = 0;
   }
}

/*******************************************************
* Reproduce durante un tiempo dado por el parámetro
* duration el tono dado por el parámetro toneToPlay
*******************************************************/
void playTone(int toneToPlay, int duration) {
   for (long i = 0; i < duration * 1000L; i += toneToPlay * 2) {
      digitalWrite(speakerPin, HIGH);
      delayMicroseconds(toneToPlay);
      digitalWrite(speakerPin, LOW);
   delayMicroseconds(toneToPlay);
   }
}

/*******************************************************
* Obtiene el tono para la nota a tocar
*******************************************************/
void playNote(char note, int duration) {
   if (note == ' '){
      delay(duration); // no tocar nota
   }
   else{
      // reproducir la nota correspondiente
      for (int i = 0; i < 8; i++) {
         if (names[i] == note) {
            playTone(tones[i], duration);
         }
      }
   }
}
```

Habíamos comentado que en Arduino existe una función para generar tonos llamada `tone()`, con ello conseguiríamos un pulso de una frecuencia y duración dadas, siendo el 50% de duración en `HIGH` y 50% en `LOW`, exactamente

lo que hacemos con la función `playTone()`... realmente no son del todo iguales ya que `playTone()` puede ejecutarse como en segundo plano, es decir si decimos que toque una nota 10 segundos, el programa seguirá ejecutándose durante esos 10 segundos, mientras que en nuestro caso el programa se bloquea y esto lo podemos ver en que en ocasiones si pulsamos el botón de parada en notas de mucha duración le cuesta reconocer la pulsación, usando `tone()` no tendríamos este problema. En contrapartida modificando pocas líneas de código nuestra función podría emitir varias notas a la vez, mientras que `tone()` solamente puede una nota a la vez aunque se aplique a distintos pines.
Hemos trabajado, pero al menos ya hemos aprendido más o menos como funciona por dentro.
La función `tone()` 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()`. Para reproducir la nota "La" durante 3 segundos sería:

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

Animo al lector a modificar el ejemplo utilizando la función `tone()` y ver las diferencias de comportamiento y como no, invito al lector a jugar con las notas y realizar sus propias "partituras".

Sintonizador de notas

Vamos a ver un pequeño ejemplo con la función `tone()` y aprovechar la peculiaridad de trabajar en segundo plano para con muy pocos componentes crear un dispositivo generador de notas (aunque le podríamos haber llamado generador de ruidos molestos). Mediante la lectura de un potenciómetro variaremos la frecuencia de las notas a reproducir de modo que podamos movernos por toda la escala de frecuencias.
Necesitaremos un potenciómetro que hará de sintonizador de frecuencia, un altavoz piezoeléctrico y aconsejo una resistencia de 1k o más para disminuir el volumen del altavoz, ya que pueden generarse ruidos molestos.
Lo que haremos será leer el valor analógico en la entrada A0. Dado que una de las patas está a 0V y la otra a 5V, la pata central actúa como divisor de tensión así que sólo tenemos que trasladar la lectura de la entrada al rango de frecuencias que queramos utilizar y reproducirlas mediante `tone()`. En el ejemplo se utilizan frecuencias entre 10Hz y 10kHz, pero podemos ajustar más el rango si queremos más granularidad del potenciómetro.

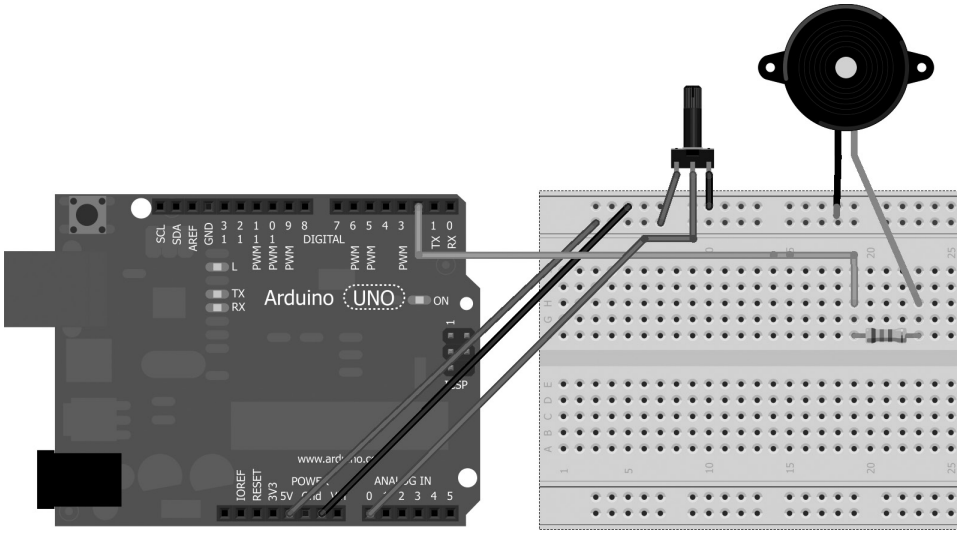


Figura 6.5. Circuito para el sintonizador de notas.

Como queremos que se esté reproduciendo constantemente las notas seleccionadas, usaremos `tone()` sin ningún parámetro de tiempos, eso quiere decir que no parará de sonar hasta que desconectemos la placa.

```
const int speakerPin = 2; // altavoz
const int potPin = A0; // potenciometro

void setup(){
   pinMode(potPin,INPUT);
   pinMode(speakerPin,OUTPUT);
}
void loop(){
   int freqRead = analogRead(potPin); // leemos el valor del potenciómetro
   int frequency = map(freqRead, 0, 1023, 10,10000); // rango de 10Hz a 10kHz
   tone(speakerPin, frequency); // reproducimos la nota
}
```

Este circuito es fácilmente ampliable con otro potenciómetro para realizar control del volumen o un botón de encendido y apagado.
Acabaremos este punto comentando una curiosidad sobre los materiales piezoeléctricos; como hemos comentado si se aplica presión en las caras del material piezoeléctrico que se encuentra en el interior del altavoz se obtiene una diferencia de potencial eléctrico entre ellas... pues bien, podemos encontrar materiales piezoeléctricos funcionando de este modo en muchas de las básculas digitales para realizar la medición del peso. Así un mismo material nos sirve para medir pesos o torsiones y para producir sonidos, simplemente cambiando la energía aplicada sobre él; curioso ¿verdad?.

Salida MIDI

Las siglas MIDI significan *Musical Instrument Digital Interface* (Interfaz Digit al de Instrumentos Musicales) y se trata de un estándar técnico que describe protocolos, conectores e interfaz digital para conectar ordenadores e instrumentos electrónicos musicales de modo que se puedan comunicar entre ellos, permitiendo hacer reproducir notas musicales, cambiar de instrumentos o registros de reproducción, composición...
El protocolo MIDI está compuesto por unas normas que no vamos a ver pero que si al lector le interesa, puede consultar en `http://www.midi.org`.
A modo genérico podríamos resumir que MIDI no transmite sonido, sino datos para que el reproductor genere las notas en el momento indicado, con las características informadas y el instrumento seleccionado. Estas instrucciones se almacenan y transmiten en valores numéricos de 0 a 127, es decir todos los datos se almacenan en bloques de 7 bits: utiliza 7 bits para seleccionar instrumento, otros 7 bits para la nota, etc... quedando el bit más significativo (el MSB) siempre a 0. Por ejemplo la guitarra española es el instrumento 24 y el violín el 40. Lo mismo pasa con las notas musicales, cada una de ellas viene dada por un valor de 7 bits resumido en la siguiente tabla.

Octava	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

El estándar MIDI incluye el conector que se debe utilizar para la comunicación y la función de cada uno de los pines durante la misma.

El conector a utilizar es de tipo DIN de 5 pines, aunque solamente se usa el pin 5 para la transmisión (por lo tanto lo tendremos que unir al pin 1 de Arduino); los pines 2 y 4 se usan para la potencia eléctrica, el pin 2 es la masa y el 4 se utiliza para la alimentación de 5 voltios, y último los pines 1 y 3 no se utilizan y están reservados para el futuro.

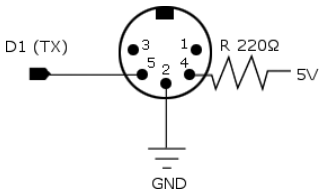


Figura 6.6. Esquema del conector DIN 5 para MIDI.

Para evitar dañar el dispositivo MIDI, introduciremos una resistencia de 220 Ω en la alimentación. Así, para el circuito necesitaremos simplemente una resistencia para limitar la corriente emitida hacia el conector MIDI y el propio dispositivo MIDI.

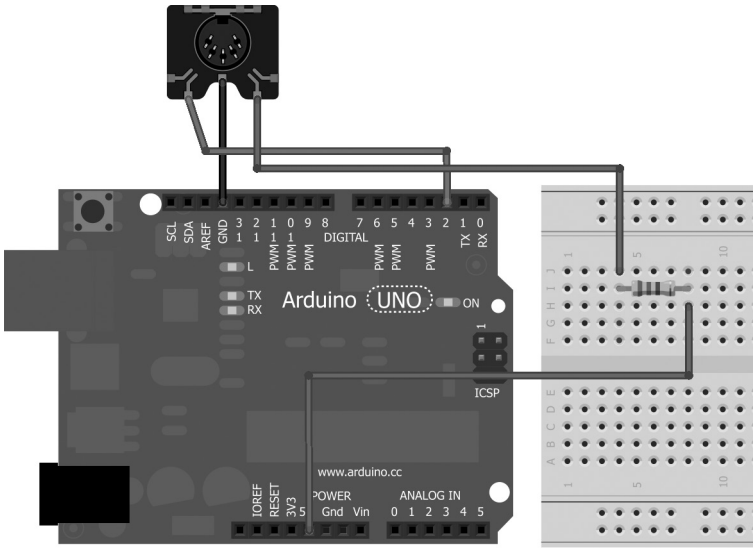


Figura 6.7. Circuito para reproducción MIDI.

Con este circuito realizaremos un *sketch* capaz de reproducir la escala musical de modo semejante a como se hizo con el altavoz piezoeléctrico, reproduciendo la escala de modo creciente y después decreciente. Pero antes de

entrar en el código del *sketch* vamos a comentar un tema más sobre MIDI y es la manera en la que reproducir una nota. El estándar MIDI tiene definidos ciertos eventos que vienen dados por un byte de estado seguido de unos bytes que serán dependientes del byte de estado. Además los dispositivos MIDI disponen de 16 canales distintos por los que reproducir la nota, pudiéndose asignar distintos instrumentos a cada canal y así poder reproducir todos los canales a la vez para generar la música. Cuando se envía un byte de estado se utilizan los 4 bits inferiores para la selección del canal y los cuatro bits superiores para la selección del evento que se quiere realizar sobre ese canal. En nuestro caso interesa el evento de reproducir una nota que según la especificación es el byte de estado 0x9-, al cual hay que añadirle el canal, por ejemplo si quisiéramos reproducir por el canal 11 enviaríamos el byte de estado 0xAB. Este byte de estado debe de acompañarse con dos bytes más (y en este orden) uno de ellos con el tono a reproducir y otro con la velocidad de pulsación de la tecla (como si fuera un piano) que realmente se traduce por el volumen al que se emitirá, si son instrumentos que no soportan velocidad, el valor por defecto es 64.

Para la liberación de la nota usaremos el byte de estado 0x8-, que tiene los mismos bytes de complemento que el 0x9-. Por último hay que tener en cuenta que la comunicación con los dispositivos MIDI se va a realizar mediante comunicación serie, y recordaremos que para iniciar la comunicación se debe introducir la velocidad en baudios que se usará durante la retransmisión; MIDI utiliza una velocidad de 31250 bps (±1% de tolerancia), por lo que así la configuraremos. En lugar de hacer la partitura en un array como lo hicimos en ejemplos anteriores, aquí lo reproduciremos directamente sobre el array de los tonos musicales ya que no compondremos ninguna canción, pero podría perfectamente ajustar para reproducir canciones como con el altavoz piezoeléctrico.

```
// notas musicales
const byte tones[8] = {60, 62, 64, 65, 67, 69, 71, 72};
const int length = 8;

void setup() {
   Serial.begin(31250);
}
void loop() {
   // reproducción creciente
   for (byte toneNumber = 0; toneNumber < 8; toneNumber++){
      playMidi(0x9B,tones[toneNumber]); // reproduce en canal 11
      delay(80);
      playMidi(0x8B,tones[toneNumber]); // libera canal 11
      delay(20);
   }
   delay(400); // pausa
```

```
   // reproducción decreciente
   for (byte toneNumber = 0; toneNumber < 8; toneNumber++){
      playMidi(0x9B,tones[toneNumber]); // reproduce en canal 11
      delay(80);
      playMidi(0x8B,tones[toneNumber]); // libera canal 11
      delay(20);
   }
}
void playMidi(byte midiEvent, byte note){
   Serial.write(midiEvent); // evento
   Serial.write(note); // nota
   Serial.write(150); // velocidad de pulsación (volumen)
}
```

Si se quisiera utilizar otro canal distinto habría que cambiar 0x9B y 0x8B por sus bytes correspondientes.

Nota:

Dado que se utiliza el pin 0 para la transmisión con el dispositivo MIDI y ocupará el puerto para la transmisión serie, se debe cargar primero el programa en la tarjeta y luego realizar la conexión hacia el dispositivo, puesto que de lo contrario nos encontraríamos con problemas en la carga del programa.

7 Sensores I

En este capítulo aprenderá a:

- Utilizar sensores de luz.
- Realizar un sensor activado por láser.
- Utilizar un micrófono.
- Conocer la posición de un dispositivo.
- Detectar vibraciones y cambios de posición.

En capítulos anteriores nuestra placa Arduino ha tenido un poco de contacto con el mundo real mediante pulsadores y potenciómetros; en este capítulo comenzaremos a realizar lecturas y medidas del mundo real de modo más automático para que nuestros circuitos puedan tener constancia de lo que pasa alrededor de ellos.

Se denominan sensores a todos los elementos que permiten captar estados o eventos físicos de modo que puedan ser transformados en variaciones de tensión o impulsos eléctricos y así poder ser interpretados dentro de nuestro circuito.

Existen multitud de sensores por ejemplo para medir temperaturas, posiciones, radiación, luz, vibración... pero todos ellos tienen en común que de algún modo varían la intensidad o tensión en alguna de las ramas del circuito eléctrico en el que se hayan conectados, por ejemplo podemos tener un sensor de movimiento implementado mediante un interruptor de mercurio; al moverse el mercurio en su interior abre y cierra el circuito y sólo tendríamos que detectar si hay electricidad en una entrada o no para saber si se está moviendo el dispositivo... es decir si hay movimiento en el sensor; del mismo modo para detectar presencia de luz existen resistencias que varían su valor dependiendo de la luz detectada.

Los sensores en Arduino los podemos encontrar en varios formatos:

- Independientes: Se tratan de unos componentes sin soporte alguno, como suelen venir las resistencias, con el elemento y sus terminales de conexión, nada más.
- Montados en placas: Muchos sensores basan sus medidas en divisiones de tensión, es decir, en la manera en la que influyen en un divisor de tensión cuando el sensor es una de las resistencias que entran en juego en dicho divisor. Para poder utilizar estos sensores deberemos de realizar un montaje junto con una resistencia externa, pero este formato de sensor, incluye las resistencias necesarias para realizar la medida. Normalmente se componen de tres terminales, siendo uno de ellos el de tensión, otro el de señal y por último otro con tierra. Otros formatos incluyen un cuarto terminal que permite usar valores umbrales para disparar valores digitales, por ejemplo podemos encontrar sensores de temperatura con cuatro terminales, dos para alimentación (5V y tierra), una para lecturas analógicas y una cuarta para digitales, de modo que cuando se supere cierta temperatura se generará una salida de 5V en ese terminal, de lo contrario será 0V. Igual que con el divisor de tensión, podemos encontrar placas que incorporen leds integrados y otros elementos configurando un circuito sencillo, que normalmente se implementa de modo externo al sensor, pero

que al venir ya preconfigurado en estas placas ahorran mucho tiempo y eliminan complejidad al circuito. Estos formatos en ocasiones incorporan un potenciómetro para regular sensibilidades o ajustar umbrales.

- 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... De la alimentación y de los pines de señal en este formato, no nos debemos de preocupar, ya que al montarlo sobre la placa Arduino, coinciden los pines de alimentación y datos, quedando nada más colocarlo, perfectamente cableado para su uso. En cuanto a los pines de señal utilizados, dependen mucho del tipo de sensores que incorpore la *shield*. Una cosa que se ha de tener en cuenta es existen *shields* destinadas para un modelo exacto de tarjeta Arduino, por lo que se debe mirar si la *shield* es compatible con la tarjeta Arduino que vayamos a utilizar, ya que entre modelos hay algunos pines que cambian de posición.

Existen infinidad de sensores y modelos distintos, en este libro se verá cómo utilizar algunos de los más comunes.

Fotorresistencia

Mediante las fotorresistencias son unos componentes cuya resistencia varía con la intensidad de luz, concretamente su resistencia disminuya cuanto mayor sea la intensidad de luz recibida. Son conocidas también como fotorresistor, sensor de luz, célula fotoeléctrica o simplemente LDR de las siglas en inglés *light-dependent resistor* (resistencia dependiente de la luz).

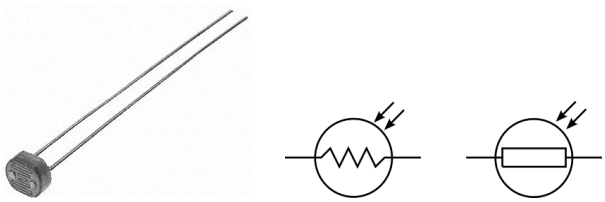


Figura 7.1. Fotorresistencia.

Comúnmente estos componentes tienen una resistencia del orden de mega ohmios (1MΩ o más) 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, tiene un retardo, y esto es debido a la naturaleza física de la

propia fotorresistencia que hace que pequeños golpes de luz puedan no ser suficientes para que varíen su resistencia de modo que sea perceptible por el circuito. En caso de necesitar un circuito que reaccione muy rápidamente a la presencia (o ausencia) de luz, se debe implementar mediante otro tipo de sensor como podrían ser sensores CMOS. Las fotorresistencias son elementos no polarizados, es decir, que no nos tenemos que preocupar de cuál de los terminales ponemos a más tensión; actúan igual que las resistencias al uso, donde no importa la polaridad.

Para trabajar con las fotorresistencias, normalmente se utiliza un montaje de división de tensión y como punto de medida, se toma precisamente la unión de la fotorresistencia con la resistencia fija según el esquema mostrado en la figura 7.2, en donde la fotorresistencia puede ser cualquiera de las resistencias, dependiendo de lo que interese en el circuito. Las fotorresistencias las podemos encontrar ya montados en pequeñas plaquitas, con la resistencia para formar el divisor de tensión incorporada.

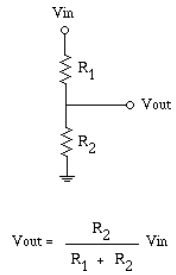


Figura 7.2. Divisor de tensión.

Para comprender mejor su funcionamiento realizaremos un pequeño circuito con una fotorresistencia y tomaremos las medidas a través de una entrada analógica mostrando su lectura en el monitor serie.

Dependiendo de la fotorresistencia que tengamos disponible, puede tener la resistencia de divisor de tensión incorporada o no, en caso de no tenerla utilizaremos una resistencia externa de 10KΩ.

Si nos fijamos en el circuito propuesto, la entrada de datos hacia Arduino se realiza sobre la caída de tensión de la fotorresistencia; cuando no haya luz tendremos que la resistencia interna de éste será de varios megaohmios, mucho mayor que los 10KΩ de la resistencia de referencia; esto nos indica que tendremos como lectura cerca de 5 voltios (al ser entrada analógica deberíamos tener valores cercanos al 1023); mientras que cuando haya luz, la resistencia de la fotorresistencia será de unos cientos de ohmios, con lo que la lectura será cercana a los 0 voltios.

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(500);
}
```

El código no tiene nada especial, pero se debe tener cuidado con el cálculo del voltaje para que pueda tener decimales; recordemos que si se hiciera `value*5/1023` nos daría el voltaje truncado por ser un cálculo entero, por eso marcamos con un .0 detrás de las constantes y conseguiremos el cálculo en coma flotante.

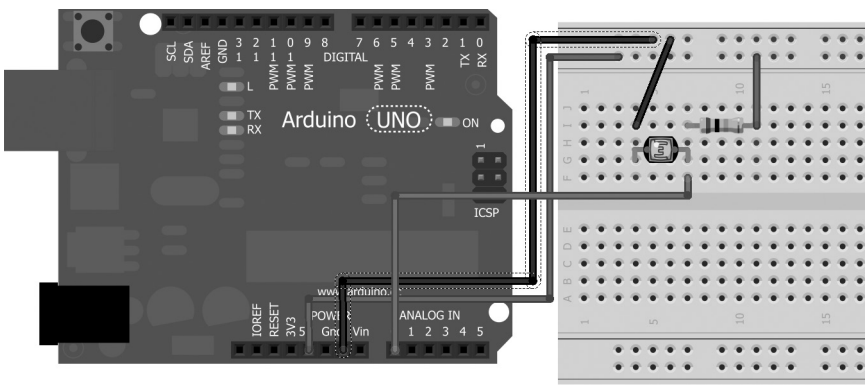


Figura 7.3. Circuito con fotorresistencia.

En el monitor serie iremos viendo la salida de la lectura analógica y su transformación a voltios.

```
Lectura : 993   Voltios: 4.85
Lectura : 970   Voltios: 4.74
Lectura : 758   Voltios: 3.70
Lectura : 531   Voltios: 2.60
Lectura : 496   Voltios: 2.42
Lectura : 657   Voltios: 3.21
Lectura : 650   Voltios: 3.18
Lectura : 555   Voltios: 2.71
```

Theremin digital

La verdad es que el último circuito puede que no nos haya dejado muy buen sabor de boca por la simplicidad y por el poco uso, así que vamos a variarlo un poco para poder darle alguna "utilidad" de manera rápida.
El theremín (o eterófono) es un instrumento musical electrónico inventado por el ruso Lev Sergeyevich Termen que consta de un par de antenas, una recta y otra curva, y se hace sonar simplemente acercando las manos (sin tocar) a dichas antenas. La antena vertical controla el tono de la nota a tocar, siendo más agudo cuanto más cerca esté la mano y la segunda controla el volumen de la misma, siendo más bajo cuanto más cerca de la antena esté. Este instrumento se basa en capacitancias producidas entre las antenas y el cuerpo del intérprete pero vamos a emular algo parecido mediante una fotorresistencia.

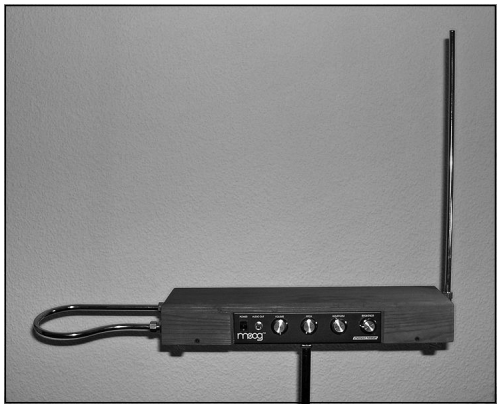


Figura 7.4. Theremín.

En el capítulo anterior se vio como realizar un sintonizador de notas mediante un potenciómetro, donde las notas iban variando dependiendo del valor que tomaba la resistencia variable del potenciómetro. En este caso usaremos una fotorresistencia para controlar esta resistencia variable y así hacer variar las notas a interpretar, de modo que al acercar la mano o alejarla de la fotorresistencia, bloquearemos o dejaremos pasar más luz y esto hará que varíe la resistencia y por ende las notas.
Para el circuito nos valdremos de una fotorresistencia, una resistencia de unos 10KΩ para el divisor de tensión de la fotorresistencia, un altavoz piezoeléctrico y optativo una resistencia, un potenciómetro u otra fotorresistencia para poner en serie con el altavoz y controlar el sonido (a mayor resistencia menor volumen).

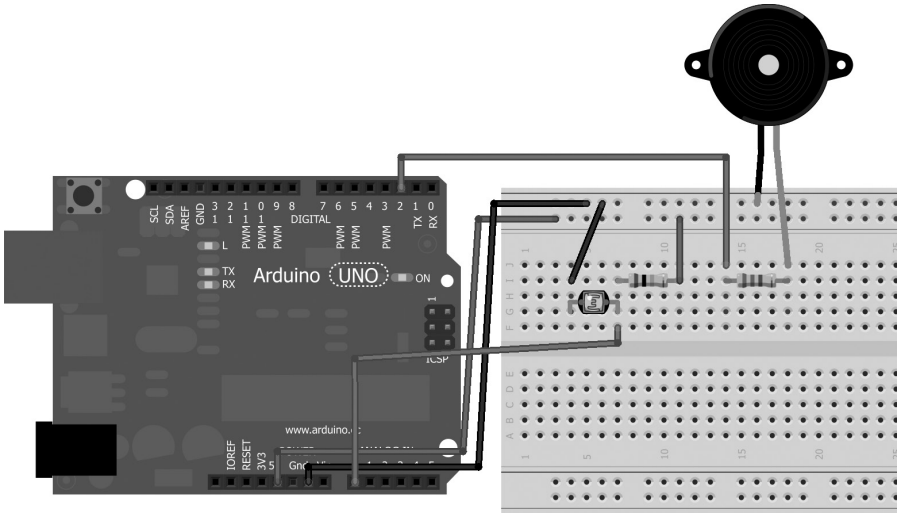


Figura 7.5. Circuito con fotorresistencia para crear theremín.

El código se debe encargar de leer los datos analógicos por la entrada A0 e ir modulando las frecuencias a reproducir usando como parámetro la lectura recibida. Modularemos las salidas entre 200Hz y 5Khz, así que sabiendo que en la entrada tendremos valores de 0 a 1023 se deberá usar la función `map()`.

```
int frequency = map(freqRead, 0, 1023, 200,5000); // rango de 200Hz a 5kHz
```

Para la salida de audio utilizaremos la función estándar `tone()`. El código completo sería:

```
const int speakerPin = 2; // altavoz
const int ldrPin = A0; // ldr
void setup() {
   pinMode(ldrPin,INPUT);
   pinMode(speakerPin,OUTPUT);
}
void loop() {
   int freqRead = analogRead(ldrPin); // leemos el valor del ldr
   int frequency = map(freqRead, 0, 1023, 200,5000); // rango de 200Hz a 5kHz
   tone(speakerPin, frequency); // reproducimos la nota
}
```

Se podría fácilmente implementar el theremín completo, es decir aquí solamente se está usando una mano para producir las notas, pero podríamos utilizar las dos y controlar también el volumen (como el theremín original) sustituyendo la resistencia en serie del altavoz piezoeléctrico por otra fotorresistencia, pero habría que separarlos bastante para que no se hicieran interferencias entre ellos y al acercar la mano a una de las fotorresistencias tapemos también la otra. Otra opción sería controlar el volumen con el potenciómetro.

Sistema de presencia con fotorresistencia

Una ventaja de las fotorresistencias es que pueden reaccionar a un rango muy amplio de frecuencias de luz, incluyendo ultravioletas e infrarrojos, lo que nos permite realizar sensores de presencia como los de los garajes.
La lectura que se ha hecho en los ejemplos anteriores sobre la fotorresistencia era analógica es decir entre 0 y 1023 para Arduino, pero también podemos tomarla como digital para detectar la presencia o no de luz basándonos en un umbral que marque la diferencia. ¿Para qué nos puede valer esto? Por ejemplo para encender las luces del patio cuando se haga de noche; no nos interesa "cuánto de noche es" sino simplemente si es de noche o no, así que podríamos implementarlo con lectura en un terminal analógico de Arduino y calcular nosotros el umbral o realizar una lectura en un terminal digital y cuando sea más de, por ejemplo, 3V la caída, será interpretado como un HIGH; en este caso el control del umbral se realizaría con la resistencia utilizada como divisor, que dependiendo de su valor, el umbral de 3V se conseguirá con más o menos luz. Si esta resistencia la implementamos con un potenciómetro, podríamos regular dicho umbral de cambio de estado digital.
Vamos a usar esta particularidad de lectura digital para realizar algo parecido a un sistema de alarma; se tratará de saber si hay o no hay luz, no de cuánta hay. El sistema se puede enfocar de dos modos distintos: colocar un emisor de luz y un objeto en medio de modo que hace sombra sobre la fotorresistencia y cuando se quita el objeto llega la luz (con lo cual haríamos que no nos robaran el objeto) o emitir un haz de luz y si algo hace sombra es que hay un objeto en medio, que podría ser el ladrón; nos centraremos en esta segunda posibilidad, pero verá el lector que es muy fácil adaptar el código del ejemplo para que trabaje como la primera opción, simplemente cambiar el `if()` de lectura de la fotorresistencia. Dependiendo de los materiales disponibles, la fuente de luz será distinta; podemos utilizar leds infrarrojos, leds normales o bien láser. Mi recomendación es usar el led láser porque es más preciso apuntando al ldr y además queda más de "de película", al crear una barrera con un haz de color.

Truco:

Si se quiere algo más preciso para detectar movimientos, existen en el mercado sensores totalmente dedicados a la detección de movimiento como los que utilizan los montajes de las alarmas comerciales y que también son compatibles con Arduino.

Para completar nuestro sistema de alarma pondremos dos botones uno para armar la alarma y otro para desarmarla. En el momento en el que se arme la alarma se debe encender el led láser para apuntar a la fotorresistencia y a partir de ese momento si deja de haber luz en la fotorresistencia y la alarma sigue armada, se debe disparar la alarma, que estará formada por un altavoz piezoeléctrico. Una vez disparada la alarma se debe esperar a que el usuario vuelva a pararla.

Para el circuito usaremos dos pulsadores, una fotorresistencia, una resistencia de 10KΩ para el divisor de tensión de la fotorresistencia, un led (a ser posible láser), un altavoz piezoeléctrico y dos resistencias de 220Ω, una para limitar la corriente en el led y otra para el altavoz, que como siempre, tanto el valor como su presencia son configurables por el lector y sirven para limitar el volumen del sonido (tengamos presente que vamos a hacer una alarma y puede molestar al resto de la gente de casa).

En caso de no tener un led láser para montajes eléctricos, se puede utilizar un puntero láser de los que se usan en presentaciones y por supuesto omitir la parte del circuito referente al led. En cuanto al código, verá que hay una sección de control del led para encenderlo y apagarlo, si se usa un puntero externo se puede dejar el código sin problemas, lo único que deberemos encender el puntero a mano.

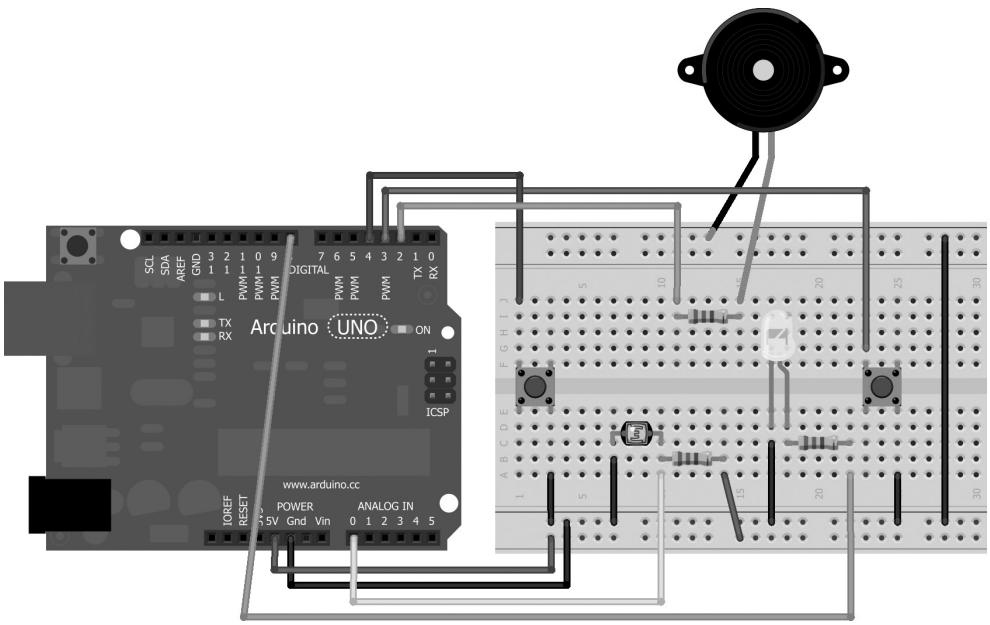


Figura 7.6. Circuito para alarma con fotorresistencia.

En el *sketch* tendremos que controlar dentro de qué estado nos encontramos, ya que puede ser que la alarma esté desarmada, armada o armada y sonando; dentro de cada uno de estos estados se deben controlar distintas entradas y salidas. Si está desarmada, nos interesa que el altavoz no produzca ruido, el led esté apagado para no consumir y esperar a que alguien apriete el botón de "armar la alarma". En caso de estar armada, la alarma debe escuchar el botón de "desarmar la alarma" y comprobar que las entradas recibidas desde la fotorresistencia son correctas (no hay nada que interrumpa el haz de luz). Por último, en caso de estar armada y la alarma sonando, se deben emitir sonidos por el altavoz y comprobar si se pulsa el botón de "desarmar la alarma". Pasemos a ver el código para realizar estas tareas.
Necesitaremos una serie de variables y constantes globales para definir los pines de entrada y salida hacia cada componente, controlar si la alarma está armada, saber si la alarma está sonando, unas constantes para definir el rango de frecuencias a las que trabajará la alarma y dos variables para controlar el efecto creciente decreciente del sonido característico de una alarma.

```
const byte speakerPin = 2; // altavoz
const byte ldrPin = A0; // ldr
const byte offAlarmPin = 3; // apagada
const byte onAlarmPin = 4; // encendida
const byte laserPin = 8; // luz
boolean isAlarmOn; // alarma armada y sonando
boolean isAlarmArmed; // alarma armada
// frecuencias máximas y mínimas a las que sonará la alarma
const int LOW_FREQ = 300;
const int HIGH_FREQ = 600;
int currentFreq = 300;
int increment = 2; // incremento de frecuencia para efecto alarma
```

Como podemos observar por las numeraciones de los pines y el esquema del circuito, el botón de la izquierda servirá para armar la alarma y el de la derecha para desarmarla. Dentro de la función `setup()` se deben configurar como entrada los pines de los botones y de la fotorresistencia, siendo de salida el del altavoz y el del led. Se inicializa también la alarma de modo que esté en estado "no armada" y sin sonar el altavoz.
En el `loop()` se deben distinguir los tres estados mencionados antes. Para conocer en qué estado estamos nos valemos de las variables `isAlarmOn` (que será verdadero si la alarma está sonando y falso si no lo está) e `isAlarmArmed` (que será verdadero si la alarma está armada y falso si no lo está). Comenzaremos la ejecución con el estado en el que la alarma está no armada.

```
noTone(speakerPin); // parar el altavoz
digitalWrite(laserPin, LOW); // apagamos el láser
if (!digitalRead(onAlarmPin)){
   isAlarmArmed = true; // armamos la alarma
```

```
   digitalWrite(laserPin, HIGH); // encendemos el láser
   delay(1000); // tiempo para cargar el ldr
}
```

Si la alarma está no armada, debemos no emitir sonido por el altavoz, para lo cual usamos la función `noTone()` sobre el pin del altavoz y el led debe permanecer apagado. La lectura debe comprobar si está pulsado el botón de armar la alarma, es decir si lo leído es `LOW`, puesto que estamos leyendo de una entrada en configuración con *pull-up*, de modo que negamos la lectura para saber en el `if ()` si está pulsado. En el caso en el que esté pulsado, se cambia de estado y se enciende el led láser. El `delay()` se ha añadido para que dé tiempo a la fotorresistencia a tomar su valor dependiendo de la luz del led; esto es porque tarda unas decimas de segundo en alcanzar el valor de resistencia inducido por la luz, no es inmediato y si no se pone el `delay()` podría saltar la alarma nada más activarla (sobre todo si estamos en ambientes muy oscuros); seguramente los buenos comerciales dirían que este retraso es para dar tiempo a que la gente salga de la sala donde pusiera la alarma.
Cuando la alarma ya se encuentre armada, se debe comprobar si la lectura del botón de "desarmar alarma" indica que está pulsado, en ese caso se cambia de estado a alarma desarmada.

```
if (!digitalRead(offAlarmPin)){
   isAlarmArmed = false; // desarmamos la alarma
   isAlarmOn = false; // paramos el sonido de la alarma
}
```

Nótese que la combinación alarma desarmada y alarma sonando no se debe dar nunca, si está desarmada tiene que estar en silencio.
Si no está pulsado el botón, se comprobará la lectura del sensor, de la fotorresistencia; en caso de ser `HIGH` significa que la fotorresistencia ha tomado un valor de ohmios alto y esto se debe a que la luz se ha bloqueado por lo que existe presencia de algo o alguien y la alarma se debe disparar.

```
if (digitalRead(ldrPin)){ // sin la luz pasa a ser high
   isAlarmOn = true;
}
```

Estando la alarma armada y disparada se debe generar salida por el altavoz a la vez que se controlan las frecuencias para generar el sonido característico de tonos crecientes y decrecientes. Se añade también un `delay()` que tiene dos funciones, la primera controlar la cadencia del sonido de la alarma y la segunda dar estabilidad; si no se pusiera las instrucciones irían tan rápido que sólo se oirían chasquidos en el altavoz. El colocar un `delay()` produce que la ejecución de las instrucciones se detenga unos instantes, pero tenerlo en esta parte del código no influye en la sensibilidad de nuestra alarma ya

que en este instante ya se ha detectado el intruso, por lo que no seguimos leyendo la entrada,;para lo único que influiría este retraso sería para la lectura del botón de desarmar.

```
if (isAlarmOn){
   tone(speakerPin,currentFreq);
   currentFreq += increment;
   // comporbar que no estamos fuera de los rangos de frecuencia
   if (currentFreq >= HIGH_FREQ){
      increment = -2;
   }
   else if (currentFreq <= LOW_FREQ){
      increment = 2;
   }
   delay(10); // efecto alarma y estabilidad
}
```

El código completo del *sketch* con la coordinación de los estados de la alarma sería éste:

```
const byte speakerPin = 2; // altavoz
const byte ldrPin = A0; // ldr
const byte offAlarmPin = 3; // apagada
const byte onAlarmPin = 4; // encendida
const byte laserPin = 8; // luz

boolean isAlarmOn; // alarma armada y sonando
boolean isAlarmArmed; // alarma armada

// frecuencias máximas y mínimas a las que sonará la alarma
const int LOW_FREQ = 300;
const int HIGH_FREQ = 600;
int currentFreq = 300;
int increment = 2; // incremento de frecuencia para efecto alarma

void setup() {
   pinMode(ldrPin,INPUT);
   pinMode(speakerPin,OUTPUT);
   pinMode(laserPin,OUTPUT);
   // comenzamos teniendo el láser apagado y la alarma no armada
   digitalWrite(laserPin, LOW);
   isAlarmOn = false;
   isAlarmArmed = false;
   // entradas de los botones
   pinMode(onAlarmPin,INPUT);
   digitalWrite(onAlarmPin, HIGH); // se activa la resistencia de pull-up
   pinMode(offAlarmPin,INPUT);
   digitalWrite(offAlarmPin, HIGH); // se activa la resistencia de pull-up
}

void loop() {
   if (isAlarmArmed){
      // si está armada debe comprobar si se pulsa el botón de parada
      if (!digitalRead(offAlarmPin)){
```

```
            isAlarmArmed = false; // desarmamos la alarma
            isAlarmOn = false; // paramos el sonido de la alarma
         }
         // se comprueba si la lectura del fotoresistor es HIGH, si es así la
         // luz está bloquada
         if (digitalRead(ldrPin)){ // sin la luz pasa a ser high
            isAlarmOn = true;
         }
         // en caso de que la alarma esté encendida generar las frecuencias
         if (isAlarmOn){
            tone(speakerPin,currentFreq);
            currentFreq += increment;
            // comporbar que no estamos fuera de los rangos de frecuencia
            if (currentFreq >= HIGH_FREQ){
               increment = -2;
            }
            else if (currentFreq <= LOW_FREQ){
               increment = 2;
            }
            delay(10); // efecto alarma y estabilidad
         }
      }
      else{
         // no está armada
         noTone(speakerPin); // parar el altavoz
         digitalWrite(laserPin, LOW); // apagamos el láser
         if (!digitalRead(onAlarmPin)){
            isAlarmArmed = true; // armamos la alarma
            digitalWrite(laserPin, HIGH); // encendemos el láser
            delay(1000); // tiempo para cargar el ldr
         }
      }
}
```

La sensibilidad de cuándo disparar, se puede controlar jugando con la resistencia del divisor; cuanto menor sea más sensible será al cambio de luz a no luz, así que podríamos poner un potenciómetro para calibrarlo.

Sensor de audio

El sensor de audio permite la captación de sonidos o golpes de aire que no tienen porqué ser en el espectro audible. De modo coloquial podemos referirnos a él como micrófono.

Pueden encontrarse en el mercado muchos tipos de sensores de audio, desde los más simples con dos terminales hasta con cuatro. Los micrófonos de dos terminales son los más difíciles de utilizar ya que para obtener un resultado óptimo hay que realizar todo un circuito a su alrededor con condensadores, transistores y algún tipo de amplificadores como el LM393. En las tiendas

se suelen encontrar sin dificultad micrófonos ya ensamblados en pequeñas placas junto con los componentes necesarios para obtener una buena recepción del sonido, incluso podemos encontrar en este formato micrófonos de alta sensibilidad para frecuencias altas. Mi recomendación es siempre que se pueda, y sobre todo al comenzar a realizar circuitos, utilizar micrófonos ya montados en placas. En caso de utilizar un sensor de audio en tarjeta con tres terminales, una de ellas se empleará para la alimentación de la tarjeta, otra como masa y la tercera como señal analógica; en caso de ser de cuatro terminales, el cuarto suele ser para salida digital, disparándose al sobrepasar un umbral de sonido. Otra de las ventajas que tienen estas placas ya montadas es que suelen disponer de otros elementos que pueden estar ligados directamente al micrófono como puede ser un potenciómetro para ajustar su sensibilidad o umbral de disparo digital, o incluso elementos no ligados como leds para indicar si está conectado o no el micrófono, o si se ha superado el valor umbral y la salida digital está en `HIGH`.

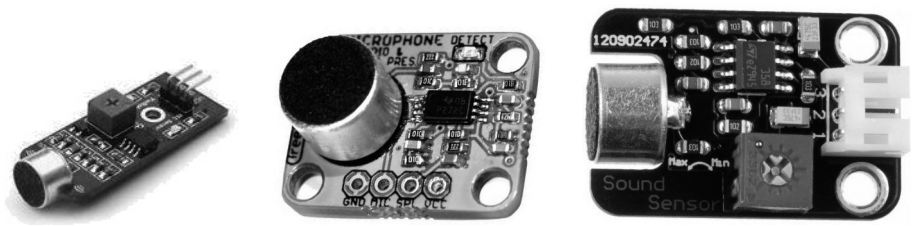


Figura 7.7. Sensores de audio.

Aunque estemos trabajando con un sensor con tres terminales, podemos utilizar el de señal analógica como disparadora de señal digital, tal y como lo hemos hecho con el circuito con la fotorresistencia.

Para ver cómo funciona el micrófono, usaremos un montaje en el que dependiendo de la fuerza de la señal recibida, se irán encendiendo distintos leds; lo que haremos será leer la señal captada y dependiendo del valor recibido encender unos leds u otros.

Para el circuito necesitaríamos un led verde, un led amarillo, un led rojo, tres resistencias de 200Ω y un micrófono; en el ejemplo usaremos un micrófono de tres terminales (5V, tierra y señal analógica).

Para el *sketch* lo que haremos será definir tres salidas para los distintos leds y unos umbrales para los cuales se activarán cada uno de ellos. En el `loop()` se procede a la lectura del valor recibido por el puerto analógico correspondiente al sensor de audio y compararlo con cada uno de los umbrales, y si es mayor que alguno de ellos se encenderá el led correspondiente.

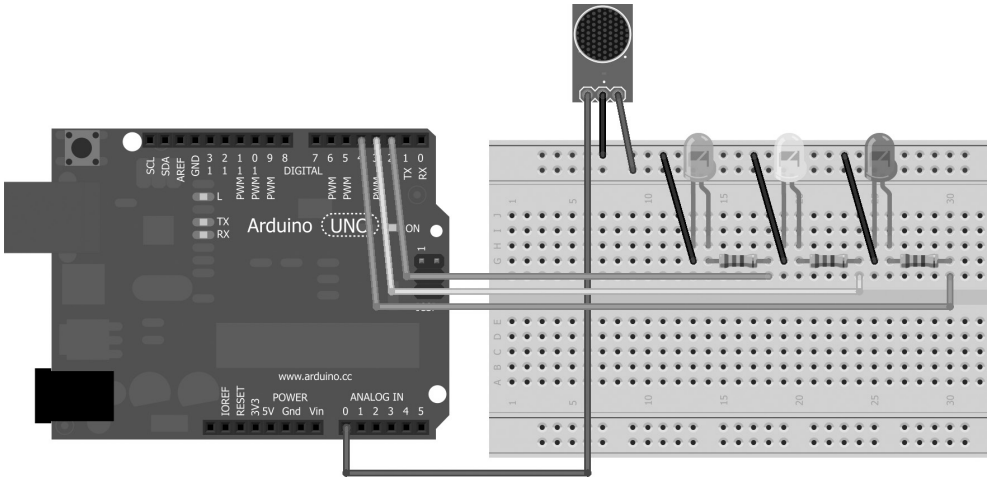


Figura 7.8. Circuito con sensor de audio.

Comenzaremos por el umbral más alto de modo que si se enciende el led rojo no se encienda el naranja ni el verde aunque el valor leído también sea mayor que el del umbral de éstos.

```
const byte audioPin = A0;
int val; // valor leido
const byte greenLed = 2;
const byte yellowLed = 3;
const byte redLed = 4;
// umbrales de los leds
const int greenThreshold = 700;
const int yellowThreshold = 850;
const int redThreshold = 1000;

void setup() {
   pinMode(greenLed, OUTPUT);
   pinMode(yellowLed, OUTPUT);
   pinMode(redLed, OUTPUT);
   pinMode(audioPin, INPUT);
}

void loop() {
   val = analogRead(audioPin);
   // se comienza a comparar por el más alto
   if(val > redThreshold) {
      digitalWrite(redLed, HIGH);
      delay(15);  // mantaner encendido
      digitalWrite(redLed, LOW);
   } else if(val > yellowThreshold) {
      digitalWrite(yellowLed, HIGH);
      delay(15);  // mantaner encendido
      digitalWrite(yellowLed, LOW);
```

```
    } else if(val > greenThreshold) {
       digitalWrite(greenLed, HIGH);
       delay(15);  // mantaner encendido
       digitalWrite(greenLed, LOW);
    }
}
```

Con este código de *sketch*, simplemente se encenderá el led correspondiente al nivel de sonido; si quisiéramos realizar un medidor de sonido, de modo que cuando sea verde se encienda sólo el verde, si es naranja se encienda naranja y verde y si es rojo los tres, simplemente deberíamos quitar los `else` dejando tres estructuras `if()` independientes de modo que si el valor es de 1020, se cumplirían los tres `if()` encendiendo los tres leds y no como ahora que sólo entra en una de las condiciones; el efecto mejora si se realiza con más leds y más niveles de lectura de entrada, consiguiendo así mayor granularidad y mejor aspecto.

Ahora que ya sabemos cómo funcionan los sensores de sonido, podemos incorporarlo a nuestra alarma añadiendo una nueva entrada y leyéndola para ver si supera un umbral y en tal caso disparar la alarma.

Si desea realizar con el micrófono es realizar reconocimiento de voz, podemos apoyarnos en librerías y software externo como BitVoicer (se puede descargar desde `http://www.bitsophia.com/BitVoicer.aspx`) para ayudarnos con la tarea.

Sensores posición y movimiento de dispositivo

A la hora de captar posiciones o bien movimientos de un dispositivo, existen múltiples implementaciones que dependiendo de cada situación, se adaptarán de mejor o peor manera.

Podemos necesitar tener conocimiento de la posición exacta de un objeto en cada momento, saber si se está moviendo, si está boca arriba o boca abajo, si ha recibido un golpe... como se puede ver son muchas las situaciones en las que podemos necesitar un sensor y para cada una de ellas hay ciertos sensores que se adaptan mejor que otros.

Para tener conocimiento de la posición de un objeto podemos utilizar giroscopios o acelerómetros, obteniendo así una posición exacta en cuanto a inclinación en cada uno de sus ejes, pero si simplemente necesitamos saber si está en posición vertical u horizontal podemos usar interruptores de mercurio. Vamos a ver algunos ejemplos de sensores de este tipo.

Acelerómetros y giroscopios

Este tipo de sensores nos permiten conocer la posición exacta del dispositivo sobre cada uno de sus ejes o su variación, pudiendo determinar la posición del dispositivo en cada momento teniendo en cuenta como eje de coordenadas el punto medio del sensor.
Lo que no nos permitirán ni el acelerómetro ni el giroscopio, es conocer la orientación del dispositivo en el mundo real; no podemos saber si se está mirando al norte o al sur (dado que eso se trata de una translación/rotación del eje de coordenadas en el que se apoya este sensor).
El acelerómetro mide la aceleración lineal en uno de los tres ejes (x, y, z); cada eje tiene su propio acelerómetro por lo que aunque pueden encontrarse acelerómetros de un solo eje o de dos ejes (con usos muy específicos de mediciones lineales), lo normal es encontrar acelerómetros de tres ejes. Son dispositivos muy baratos y su uso con Arduino es muy sencillo. Los preparados para ser usados en prototipos suelen tener cinco patillas: una de tensión, una de tierra y tres para proceder con los datos de las lecturas de los acelerómetros, una independiente por cada eje. Existen de diferentes magnitudes de medida; se debe tener en cuenta la máxima aceleración a la que se le puede someter (no es lo mismo un acelerómetro para un fórmula 1 que para un coche a pilas teledirigido).
Las aceleraciones se miden en G, (1G corresponde a la aceleración por atracción terrestre 9.8066 m/s^2) y los acelerómetros se seleccionan por el número máximo de G soportado. Por ejemplo si seleccionamos un acelerómetro de 2G y nuestra entrada en Arduino proporciona 10 bits de resolución tenemos una precisión de 2 / 1024 = 0.002G mientras que si lo seleccionamos de 20G tendríamos una precisión de 0.02G, mucho menor que en el acelerómetro de 2G, pero si nuestro acelerómetro va a ir montado en un dispositivo capaz de generar aceleraciones de 15G, al poner uno de 2G daría siempre de lectura 1024, por lo que hay que seleccionar bien para tener una precisión alta pero vigilando los valores máximos.

Advertencia:

Es muy importante leer las hojas del fabricante sobre el sensor que se vaya a utilizar, ya que podemos dañar el sensor si se cablea mal o se dan valores de tensión mayores de los que indica el fabricante para usar como voltaje referencia o alimentación; algunos sensores necesitan alimentaciones de 3V por lo que si usáramos 5V podríamos estropear el sensor.

Los giroscopios en cambio miden velocidades angulares en α, β, γ y son los sensores más indicados para sistemas de estabilización y cambios de orientación. A diferencia de los acelerómetros, los giroscopios miden cambios; no tienen una referencia fija. Para conocer el giroscopio que se adapta a nuestras necesidades, se debe conocer el máximo cambio de grados por segundo que tendremos en nuestro dispositivo. Del mismo modo que con los acelerómetros debemos tener cuidado de seleccionar un giroscopio con la suficiente precisión pero sin que nos quedemos cortos en la máxima variación de grados por segundo, y es que no es la misma velocidad de giro la de un taladro que la de un carrusel de feria.
Aunque se pueden encontrar tanto acelerómetros como giroscopios por separado, lo normal actualmente es encontrarlos integrados en una misma placa denominada IMU (*Inertial Measurement Unit*, unidad de medida inercial). Su capacidad de medición viene dada por el número de grados de libertad, sumando el número de ejes que leen los acelerómetros más el número de velocidades angulares que leen los giroscopios.

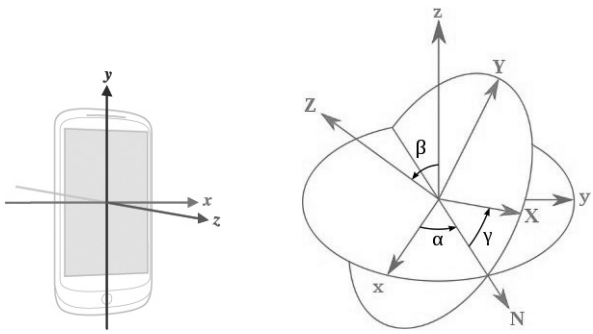


Figura 7.9. Ejes y ángulos de acelerómetros y giroscopios.

Ya que los datos de lectura de estos sensores se ven influidos por parámetros externos como la temperatura, podemos encontrar en el mercado algunas IMU con sensor de temperatura, presión... para mejorar su medición. Sin querer entrar más en detalle, un problema que presentan estos sensores es que los microprocesadores tardan un tiempo en procesar los datos leídos, y entre lectura y lectura, podemos perder valores; existen varios métodos matemáticos para la corregir estas lecturas perdidas aunque el más utilizado es el conocido como filtro Kalman.
El uso de los acelerómetros es casi directo, no se necesita ningún elemento adicional (como delimitadores de corriente), simplemente se debe alimentar y seleccionar cada una de sus entradas como entradas analógicas.

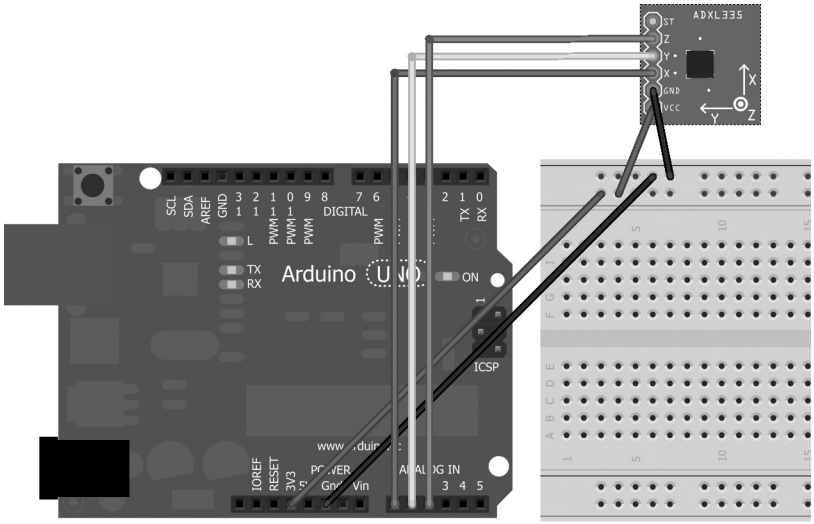


Figura 7.10. Circuito con acelerómetros.

Cada modelo tiene su propia numeración de patillas e incluso distinta posición por lo que hay que prestar atención cuando se monta. En Arduino, unos de los acelerómetros más utilizados son los de la serie ADXL3xx de Analog Devices, que son muy fáciles de cablear y baratos. Ojo, estos dispositivos suelen alimentarse a 3.3V en lugar de a 5V.

Para obtener sus datos en el *sketch* nos valdría con una lectura directa sobre los puertos de entrada analógicos y luego procesarlos dependiendo de las escalas medidas por el acelerómetro:

```
void loop(){
   Serial.print("Valor x: ");
   Serial.println(analogRead(A0));
   Serial.print("Valor y: ");
   Serial.println(analogRead(A1));
   Serial.print("Valor z: ");
   Serial.println(analogRead(A2));
   delay(100);
}
```

En cuanto a los giroscopios, se actúa del mismo modo que con los acelerómetros, se obtiene su lectura y se debe procesar dependiendo de las escalas de medida del giroscopio para tener la magnitud real y teniendo en cuenta su estado de reposo. Hagamos un ejemplo; si se tiene un giroscopio con una sensibilidad de 3.33mV/grados/s y se quiere obtener su lectura real en Arduino se debe aplicar la fórmula:

```
velodidadGiro = (lecturaGiro-ceroGiro)/sensibilidadGiro
```

Donde la `lecturaGiro` es la lectura analógica que hemos realizado, en nuestro caso como tenemos un ADC (*Analog to Digital Converter*, convertidor analógico digital) de 10 bits, tendremos un valor entre 0 y 1023; `ceroGiro` es el valor que tendría la entrada si el giroscopio estuviera en reposo, que se configura en el `setup()` y normalmente suele ser la mitad del valor total (serían 2.5 voltios, 511 en lectura analógica); por último la `sensibilidad-Giro` es la sensibilidad marcada por el fabricante, en nuestro caso los 3.33mV/grados/s, pero se debe transformar en "lectura analógica", es decir, cuántos bits de entrada cambian al cambiar un 3.33mV en la entrada. Para conocer este valor necesitamos saber el valor del voltaje de referencia del giroscopio, que se puede utilizar alguno de los disponibles en la placa Arduino o externos (mediante la función `analogReference()` vista anteriormente) si seleccionamos 5V (habría que revisar si el sensor soporta esta tensión) sería:

```
sensibilidad/5*1023 = 0.00333/5*1023= 0.6813
```

Si la tensión referencia fuera menor para obtener mayor precisión, por ejemplo 2.5 voltios la fórmula sería:

```
0.00333/2.5*1023= 1.3626
```

Con esta sensibilidad ya podemos calcular la velocidad de giro mediante la fórmula anterior:

```
velodidadGiro = (lecturaGiro-ceroGiro)/sensibilidadGiro =
   (lecturaGiro-511)/1.3626
```

Si además queremos saber la posición en la que se encuentra, solo hay que añadir a la posición anterior el resultado de multiplicar la `velocidadGiro` calculada y que viene en grados/s por el tiempo que hace que se tomó la posición anterior. En la figura 7.11 se puede ver cómo se cablearía un IMU (acelerómetros + giroscopio) de 5 grados de libertad (x, y, z, α, β), es decir, tres acelerómetros y dos giroscopios.

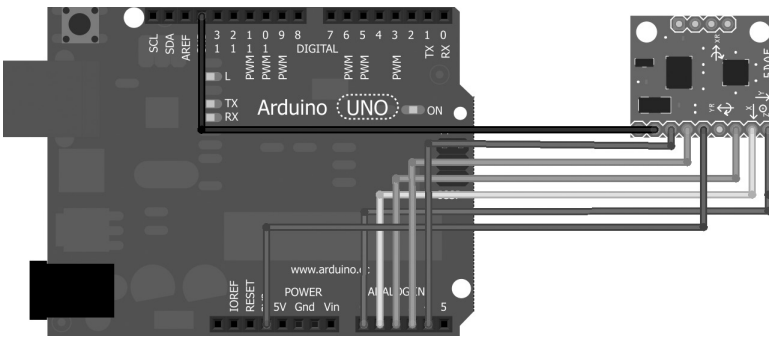


Figura 7.11. Circuito con IMU de 5 grados de libertad.

Existen IMUs donde en lugar de tener un terminal para cada una de las mediciones, disponen de una sola salida y unas entradas que van a un multiplexor para seleccionar la lectura que se desea realizar; en estos casos más que nunca es importante leerse el manual de uso del fabricante para saber los valores que aplicar al multiplexor para obtener cada lectura.

Sensores posición, vibración e inclinación

Es posible que no necesitemos tanta precisión en el conocimiento de la posición del dispositivo y que simplemente queramos saber si está horizontal, o si está inclinado, o si se está moviendo... pero no cuántos grados de inclinación tenemos. Para ello podemos usar otro tipo de sensores más económicos y fáciles de utilizar, como pueden ser los interruptores de mercurio. Este tipo de interruptores son unos elementos muy baratos que consisten en unos recipientes estancos donde hay dos terminales con separación entre ellos; dentro de esos recipientes existe también una bolita de mercurio que se mueve libremente y al pasar por encima de los dos terminales cierra el circuito trabajando como interruptor. El funcionamiento básico es el explicado, pero los encapsulados disponibles son muchos y variados; por ejemplo existen algunos interruptores de mercurio que tienen los dos terminales en el fondo y se cierran el circuito al poner el elemento en vertical, otros los tienen en el medio y solo se cierra el circuito cuando está totalmente horizontal, existen que tienen unos bornes en un lado y otros en el otro lado y dependiendo de hacia qué lado esté inclinado, abrirá o cerrara cada uno de los circuitos... hay multitud, los hay incluso que vienen encapsulados con un led para saber si está en circuito abierto o cerrado.
Dada la multitud de opciones, podemos buscar aquella que más se aproxime a nuestras necesidades y ajustar la inclinación de las patillas o del recipiente de mercurio para acabar adaptar el cierre de circuito (o apertura) a la inclinación deseada en nuestro montaje.

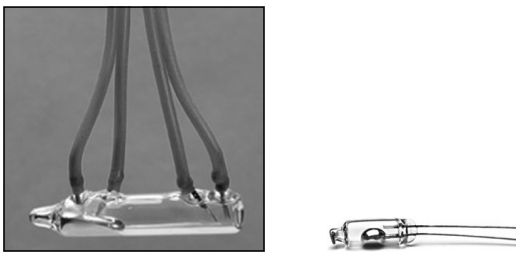


Figura 7.12. Interruptores de mercurio.

Para detectar pequeñas vibraciones podemos encontrar una variación de los interruptores de mercurio que son los sensores de vibración; unos elementos semejantes pero que en su interior tienen una bolita metálica que se desplaza, haciendo contacto con los bornes en las vibraciones. Si a estas soluciones se las dota de inclinación en el recipiente, se pueden utilizar como sensores de inclinación. Otra implementación de estos sensores de vibración se realiza mediante un tubo conductor que a su vez es uno de los bornes y en su interior existe un muelle con uno de sus terminales libre que hará la función de segundo borne; al detectar vibraciones el muelle se desplaza tocando el tubo y cerrando el circuito.
Para usar estos sensores en nuestros montajes, simplemente debemos hacer que cierren o abran un circuito y leer con Arduino el estado de dicho circuito como en la figura 7.13.

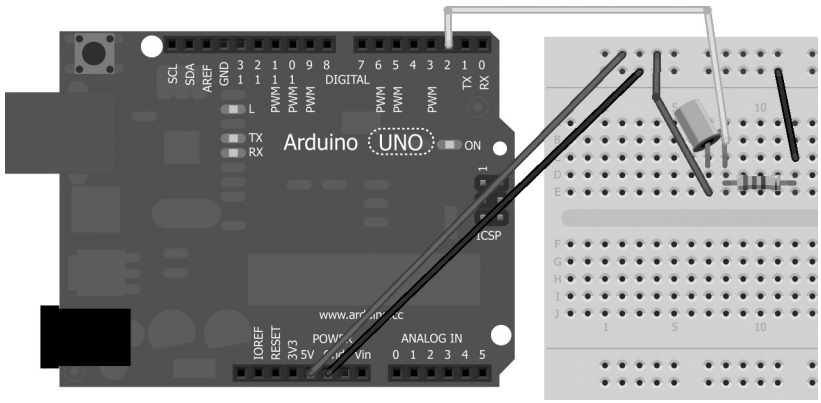


Figura 7.13. Circuito con sensor de inclinación.

El *sketch* simplemente se encargaría de leer la entrada en busca de valores `HIGH` para circuito cerrado y `LOW` en abierto.
Ahora que ya conocemos unos pocos sensores más podemos mejorar nuestra alarma añadiendo un control volumétrico para que cuando se capte un sonido que supere cierto umbral se dispare la alarma. Para ello incluiríamos un micrófono del cual se leerá su entrada que se cotejará con el umbral máximo configurado; si lo supera debe encenderse la alarma. Estas comprobaciones solamente se deben realizar si la alarma está armada. El código sería algo semejante a:

```
val = analogRead(audioPin);
if (val > soundThreshold){ // si supera umbral
   isAlarmOn = true;
}
```

Para evitar que nos puedan robar la alarma (sería el colmo), añadiremos unos acelerómetros, los cuales se leerán en búsqueda de variaciones en sus valores. Se podría utilizar un sensor de vibración en lugar de usar un acelerómetro, pero si se moviera muy poco a poco la alarma es posible que no saltara el sensor de vibración, así que procederemos con los acelerómetros para tener mayor seguridad. Lo que haremos es que al armar la alarma, se lean los valores de los acelerómetros y una vez armada la alarma se vayan leyendo los valores de estos; si hay una diferencia mayor del umbral marcado, la alarma debería saltar. A la hora de armar la alarma salvamos los valores actuales:

```
accelX = analogRead(XPin);
accelY = analogRead(YPin);
accelZ = analogRead(ZPin);
```

Cuando la alarma está armada, se comprueban las nuevas lecturas por si superan el umbral permitido.

```
if ( abs(accelX - analogRead(XPin)) > accelThreshold ||abs(accelY -
      analogRead(YPin)) > accelThreshold ||abs(accelZ - analogRead(ZPin)) >
      accelThreshold   ){
   isAlarmOn = true;
}
```

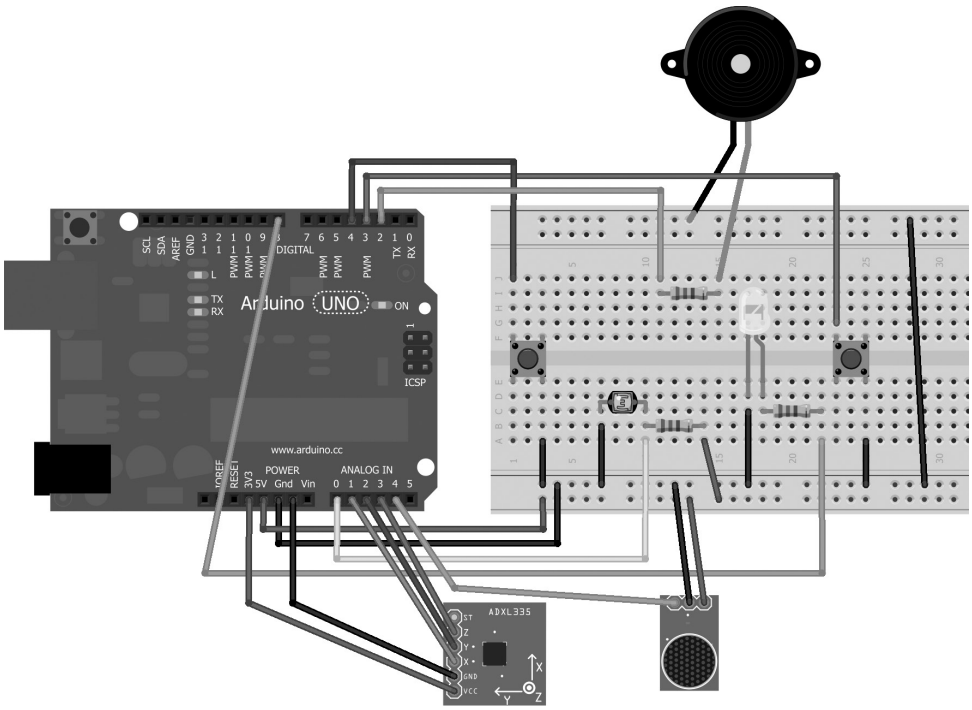


Figura 7.14. Circuito de alarma con sensor de movimiento y sonido.

Si cualquiera de los valores supera el umbral, hay que disparar la alarma; es por lo que se ha hecho un *or* lógico de cada una de las comprobaciones. La función `abs()` nos permite obtener el valor absoluto de la lectura, ya que el acelerómetro puede tener diferencias negativas o positivas frente al valor obtenido durante la puesta en marcha de la alarma.
Incluiremos estos dos nuevos sensores en el *sketch* de la alarma añadiendo también las variables necesarias para su correcto funcionamiento.

```
const byte speakerPin = 2; // altavoz
const byte offAlarmPin = 3; // apagada
const byte onAlarmPin = 4; // encendida
const byte laserPin = 8; // luz
const byte ldrPin = A0; // ldr
const byte XPin = A1; // acelerómetro x
const byte YPin = A2; // acelerómetro y
const byte ZPin = A3; // acelerómetro z
const byte audioPin = A4; // micrófono

const int soundThreshold = 500;  // máximo sonido permitido
const int accelThreshold = 100;  // máxima variación en acelerómetros
                                 // permitida

// valores iniciales de los acelerómetros
int accelX = 0;
int accelY = 0;
int accelZ = 0;

boolean isAlarmOn; // alarma armada y sonando
boolean isAlarmArmed; // alarma armada

// frecuencias máximas y mínimas a las que sonará la alarma
const int LOW_FREQ = 300;
const int HIGH_FREQ = 600;
int currentFreq = 300;
int increment = 2; // incremento de frecuencia para efecto alarma

void setup() {
   pinMode(ldrPin,INPUT);
   pinMode(speakerPin,OUTPUT);
   pinMode(laserPin,OUTPUT);
   // comenzamos teniendo el laser apagado y la alarma no armada
   digitalWrite(laserPin, LOW);
   isAlarmOn = false;
   isAlarmArmed = false;
   // entradas de los botones
   pinMode(onAlarmPin,INPUT);
   digitalWrite(onAlarmPin, HIGH); // se activa la resistencia de pull-up
   pinMode(offAlarmPin,INPUT);
   digitalWrite(offAlarmPin, HIGH); // se activa la resistencia de pull-up
}

void loop() {
```

```
    if (isAlarmArmed){
       // si está armada debe comprobar si se pulsa el botón de parada
       if (!digitalRead(offAlarmPin)){
          isAlarmArmed = false; // desarmamos la alarma
          isAlarmOn = false; // paramos el sonido de la alarma
       }
       if (!isAlarmOn){
          // se comprueba si la lectura del fotoresistor es HIGH, si es así
          // la luz está bloquada
          if (digitalRead(ldrPin)){ // sin la luz pasa a ser high
             isAlarmOn = true;
          }

          // sonido
          if (analogRead(audioPin) > soundThreshold){  // si supera umbral de
                                                       // sonido
             isAlarmOn = true;
          }

          // acelerómetros
          if ( abs(accelX - analogRead(XPin)) > accelThreshold ||abs(accelY
                - analogRead(YPin)) > accelThreshold ||abs(accelZ -
                analogRead(ZPin)) > accelThreshold   ){
             isAlarmOn = true;
          }
       }
       // en caso de que la alarma esté encendida generar las frecuencias
       if (isAlarmOn){
          tone(speakerPin,currentFreq);
          currentFreq += increment;
          // comporbar que no estamos fuera de los rangos de frecuencia
          if (currentFreq >= HIGH_FREQ){
             increment = -2;
          }
          else if (currentFreq <= LOW_FREQ){
             increment = 2;
          }
          delay(10); // efecto alarma y estabilidad
       }
    }
    else{
       // no esta armada
       noTone(speakerPin); // parar el altavoz
       digitalWrite(laserPin, LOW); // apagamos el laser
       if (!digitalRead(onAlarmPin)){
          // obtenemos los valores de los acelerómetros
          accelX = analogRead(XPin);
          accelY = analogRead(YPin);
          accelZ = analogRead(ZPin);
          isAlarmArmed = true; // armamos la alarma
          digitalWrite(laserPin, HIGH); // encendemos el laser
          delay(1000); // tiempo para cargar el ldr
       }
    }
}
```

Como se puede observar, se ha retocado la parte del código correspondiente a cuando la alarma está armada para que no compruebe los sensores si la alarma está sonando (ya está sonando da igual la lectura de los sensores). Esto se ha hecho porque al contrario que en el ejemplo anterior de la alarma, en este hay muchas lecturas a sensores que ocupan muchos ciclos del microprocesador y son lecturas que no sirven para nada (recordemos que la alarma está sonando), por lo que si se encuentra sonando salta estas lecturas.

```
if (!isAlarmOn){ ...}
```

Durante las comprobaciones, si hubiera saltado la alarma por la fotorresistencia, se estarían haciendo lecturas inútiles al altavoz y a los acelerómetros durante un ciclo; se podría retocar el código para que no las hiciera pero queda mucho más legible así y es una mejora mínima puesto que solo afecta al ciclo en cuestión.

Una pega a esta alarma es... si la alarma se consigue mover sin modificar la posición absoluta de sus ejes la alarma no sonaría, dicho de otra manera si está en horizontal y la conseguimos trasladar en horizontal, la alarma no sonaría puesto que lo que se realiza es una traslación de los ejes del dispositivo (relativa a otros ejes) con lo que este tipo de movimientos no son captados por los acelerómetros. Necesitaríamos un sensor que utilice otros ejes externos a los del dispositivo como podría ser un GPS, donde notaríamos que lo están desplazando (pero no notaríamos si lo giran), aunque es casi imposible mover el dispositivo sin modificar su posición, aunque sea un poco y este poco lo podemos controlar con la constante `accelThreshold`.

8 Sensores II

En este capítulo aprenderá a:

- Utilizar sensores de temperatura.
- Obtener la humedad del ambiente.
- Configurar y leer las posiciones de un joystick.
- Controlar un programa de ordenador mediante un joystick.

En el capítulo anterior comenzamos a ver algunos de los sensores más utilizados; en este capítulo continuaremos descubriendo nuevos sensores que más adelante podremos incorporar a nuestros montajes.

Sensor de temperatura

Los sensores de temperatura, como su propio nombre indica, nos permitirán conocer la temperatura del entorno en el que se encuentre el sensor. Las temperaturas medidas siempre son referidas a una escala; cuando decimos que hace 20 grados centígrados, nos referimos a la escala centígrada, pero existen otras muchas que marcan su origen y a partir de él se rigen de ciertos factores que permiten medirla temperatura. El origen de medida no es el mismo para todas las escalas, como podría pasar en unidades de medida de distancia, ya que por ejemplo da igual si son pies, yardas o kilómetros que 0 siempre será 0; en temperatura no existe un 0 universal, y lo marcan las propias escalas.
Existen muchas escalas como la Reaumur, la Rankie, la Romer, pero las más utilizadas en la vida cotidiana son:

- La centígrada o grados 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. Es la que se utiliza comúnmente en Europa.
- La Fahrenheit: Usada por unos pocos países (Lo malo es que entre ellos está EEUU), tiene el 0 de la escala en la temperatura de congelación de una solución salina que parece ser usó Daniel Fahrenheit cuando diseño su escala (existen varias versiones de cómo acaeció). La temperatura a la que se congela el agua en esta escala son los 32 grados Fahrenheit y hierve a 212 °F, si tomamos la diferencia de temperaturas y lo dividimos entre 180 obtenemos el incremento de temperatura que representa un grado Fahrenheit.
- La Kelvin: Basada en la escala Celsius; el origen de coordenadas lo marcan los 0 grados Kelvin, temperatura denominada 'cero absoluto' y corresponde al punto en el que las moléculas y átomos de un sistema tienen la mínima energía térmica posible. Cada grado Kelvin corresponde térmicamente a un Celsius. El 0 Kelvin son los −273.15 °C. Esta es la escala del sistema internacional de medidas para temperatura.

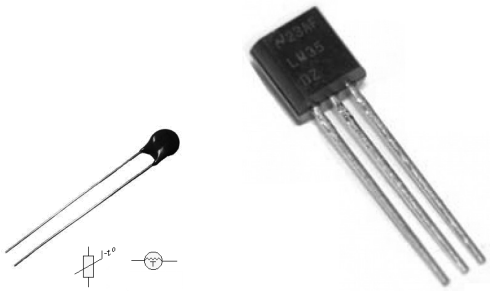


Figura 8.1. Termistor y sensor LM35.

Una de las maneras de medir la temperatura es mediante termistores que son unas resistencias que varían su valor dependiendo de la temperatura a la que se encuentren. Si se opta por utilizar este sistema se debe realizar un divisor de tensión, tomar la lectura en el punto medio del mismo (como se hizo con la fotorresistencia) y calibrarlo con la resistencia del divisor para conocer la temperatura medida dependiendo de la tensión obtenida. Otra opción mucho más práctica es utilizar sensores de temperatura ya empaquetados que no necesitan de resistencias externas para obtener el valor; estos sensores no nos darán la temperatura en ninguna de estas escalas, sino que nuevamente obtendremos un valor analógico que tenemos que encargarnos de transformar en la escala que nos interese. Los sensores suelen fabricarse de modo que sean lineales a los grados Celsius o a los grados Fahrenheit, de manera que un incremento de un grado en una de estas escalas genere un incremento de tensión proporcional, pero siempre se puede transformar la temperatura entre distintas escalas mediante fórmulas matemáticas. Antes de disponerse utilizar un sensor de medición de temperatura, se deben mirar los papeles del fabricante para conocer el rango de temperaturas que puede medir, teniendo en cuenta las escalas.

Unos de los sensores de temperatura más utilizados son los LM34 y LM35; el primero de ellos es lineal para temperaturas medidas en grados Fahrenheit, mientras que el segundo es lineal para medidas en grados Celsius.

Para ver un ejemplo de cómo funcionan usaremos un LM35D que es muy fácil de conseguir. Existen múltiples variantes como el LM35A, LM35CZ o el LM35DT que también podemos utilizar ya que su uso es muy similar, pero en este caso se deberían tener en cuenta que los rangos de temperaturas que miden son distintos.

Realizaremos un circuito en el que leeremos la temperatura y la mostraremos en el monitor serie; para realizar el montaje simplemente se necesita un LM35 o algún sensor de temperatura semejante.

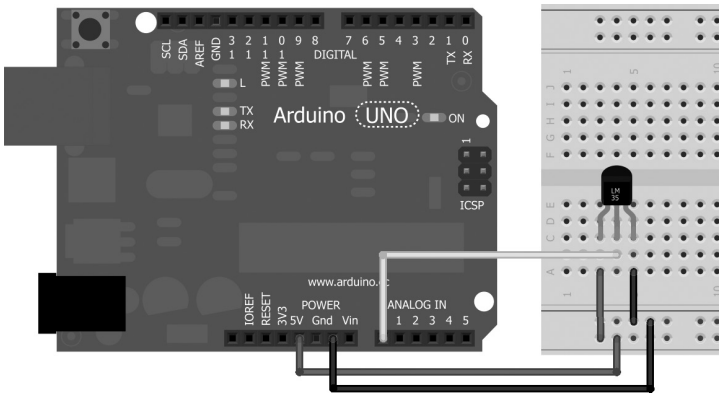


Figura 8.2. Circuito con sensor LM35.

En el *sketch* debemos leer la entrada analógica y obtener la temperatura representada por ésta, pero no es directo. Lo primero es que el LM35 tiene como salida máxima 1V, y si usamos de tensión de referencia del ADC los 5V como hemos estado haciendo hasta ahora quiere decir que estaremos desperdiciando el 80% de los valores posibles, dado que nunca tomará valores superiores a 1V; lo que haremos es trabajar con la tensión de referencia interna que es de 1.1V, así sólo desperdiciaremos aproximadamente un 9% de los valores posibles, ganando así precisión (lo ideal sería trabajar con una referencia de 1V). Sabiendo que podemos tener un rango de valores de voltaje de 0 a 1.1V y que estos se representan mediante un 10 bits, tenemos 1024 valores posibles, o lo que es lo mismo 1.1V/1024 valores = 0.00107421875V = 1.074mV por valor, es decir podemos hacer incrementos en la lectura de 0.001074 voltios; si trabajáramos a 5 voltios de referencia tendríamos 5V/1024 posiciones = 0.0048828125V = 4.88mV por valor, lo que hace que tengamos mucha menor precisión. Sigamos pues con el cálculo para 1.1V. Según las especificaciones del fabricante este sensor mide de 0° centígrados a 100° centígrados de modo lineal con un factor de escala de +10mV/°C; si 10mV son 1°C y cada valor de la entrada analógica para 1.1V representa una variación de 1.074mV, quiere decir que para representar un grado centígrado necesitaremos 10mV/1.074mV = 9.31 valores en la entrada. Este es un número mágico, ya que la entrada leída dividida por este número nos da directamente la temperatura obtenida en grados centígrados.
Con este valor mágico ya podemos proceder a efectuar el *sketch* para el sensor de temperatura.

```
float tempC; // temperatura en celsius
int analogValue; // valor leido 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 por cada grado centígrado, que tal y como hemos calculado anteriormente es 9.31. Ya podemos ver la temperatura en nuestro monitor serie.

Sensor de temperatura y humedad

Los sensores de temperatura pueden obtenerse por separado u obtenerse acompañados de otros sensores que complementen su función.
Muchos sistemas se ven afectadas sus condiciones de trabajo por temperaturas, presiones y humedades, por lo que es muy normal encontrar estos tres sensores encapsulados; también podemos encontrar otras combinaciones como temperatura y presión, presión y humedad o bien temperatura y humedad.
De este último sensor existen dos modelos de muy bajo coste que están muy extendidos, se trata del DHT11 y el DHT12. Aunque son bastante parecidos, el DHT11 es un poco más barato porque tiene menor precisión y menor rango de medida y lo podemos encontrar suelto o como parte de algún empaquetado como el HM2301. Utilizaremos uno de ellos para realizar una pequeña estación meteorológica.

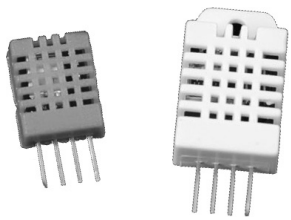


Figura 8.3. Sensores DHT21 y DHT22.

Su conexión es muy sencilla, se utilizan tres de los cuatro terminales (quedando el tercero libre, si es que estamos en la versión de cuatro terminales); uno para la alimentación, otro para tierra y otro para la señal. Quizá en un principio pensáramos que se iba a utilizar un terminal para la temperatura y otro para la humedad, de modo como se hizo con el LM35, pero no es así.

Este sensor tiene la particularidad de que simplemente tiene un terminal para transmitir tanto la temperatura como la humedad, por lo que utiliza una transmisión serie de los datos sobre dicho terminal. Para ser capaces de leer la información, debemos comunicarnos con el sensor de manera que nos podamos entender, y esta forma la dicta el fabricante.

Cuando se transmiten datos sobre una misma pata se hace de forma que la información debe ir codificada y en cada momento tanto el emisor como el receptor deben de saber de qué dato transmitido se trata; lo que se suele hacer es comenzar la transmisión con lo que se llama un *handShake* (choque de manos), es decir el sistema cliente (en nuestro caso Arduino) le envía una cadena de ceros y unos en un orden establecido y con una duración establecida, para que el sensor sepa que debe comenzar a enviar los datos; los datos son enviados también siguiendo una pauta establecida por el protocolo utilizado para la comunicación. La transmisión de los datos en este caso se realiza en 40 bits (según fabricante) que se transmitirán de la siguiente manera, 50 µs en estado `LOW` indica comienzo de un bit a transmitir y si los siguientes 28 µs son en `HIGH` indica que se envía un 0 pero si está en `HIGH` 70 µs significa que lo que se transmite es un 1; para la transmisión del siguiente bit vuelve a tener una salida `LOW`.

Los 40 bits son 5 bytes 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.

Las especificaciones sobre este sensor (o alguno compatible) pueden consultarse en Internet (`http://www.electrodragon.com/w/images/6/6f/DHT21.pdf`).

Usando el DHT21 haremos un montaje para mostrar la humedad y temperatura mediante el monitor serie; para el circuito necesitaremos un DHT21 y una resistencia de 10kΩ; si se quiere mayor estabilidad se puede añadir un pequeño condensador de unos 100nF entre la alimentación y tierra, pero es optativo.

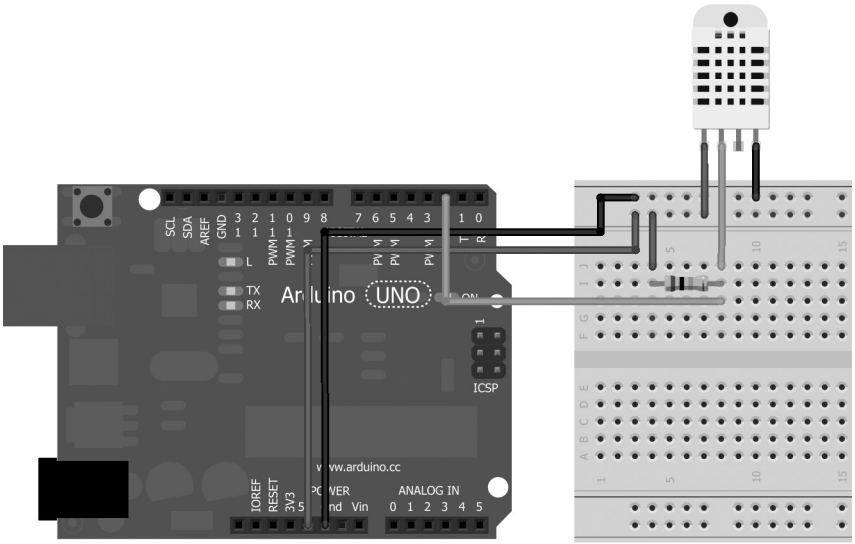


Figura 8.4. Circuito con DHT21.

Por lo que hemos visto de las características del sensor, lo que deberemos hacer en el *sketch* es comenzar la transmisión de los datos del sensor hacia la placa Arduino enviando desde ésta la secuencia de comienzo de transmisión, y una vez enviada comenzar a 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.

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

En el `setup()` configuraremos el pin 2 como de salida y en `HIGH`, dado 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 de eso 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:

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

Una vez recibidos todos los bytes debe comprobarse 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 50µs y luego se pasa a `HIGH`, si la duración de `HIGH` es de entre 26 y 28µs quiere decir que es un 0, si es de 70µs quiere decir que es un 1; lo que haremos será por cada bit, leer mientras esté a `LOW`, cuando pase a `HIGH` esperar 30 µs y si la lectura es nuevamente `HIGH` quiere decir que es un 1 (ya que hemos pasado los 28µs) y si no será un 0; una vez obtenido el valor, lo almacenamos y luego 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);
}
```

En esta función hemos usado un par de 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, el código completo del *sketch* quedaría así:

```
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 condicion 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;
}
```

Para probar los errores, valdría con desconectar el cable del pin 2 y nunca se llevaría a cabo el *handshake*, propiciando un error.
Supongo que el lector habrá encontrado este ejemplo un poco más complejo que los anteriores; no se asuste, la mayor parte de componentes que resultan complejos de programar tienen librerías disponibles para facilitar la tarea; por ejemplo para los sensores DHT podríamos utilizar la librería disponible en GitHub (`https://github.com/adafruit/DHT-sensor-library/archive/master.zip`). Al utilizar esta librería el código del *sketch* se ve drásticamente reducido. Si adaptamos el ejemplo disponible con la propia librería a nuestro caso, el *sketch* sería:

```
#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.");
   }
}
```

Mucho más reducido ¿verdad? Un par de aclaraciones sobre este código. Lo primero que nuevo es la línea:

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

con ello se indica que se va a utilizar la librería DHT (la tenemos que tener accesible en nuestro *sketch*, para ello hay que leerse el fichero que viene para su instalación). Dentro de ese fichero se encuentran definiciones de constantes y de la clase `DHT`; se trata de un archivo de cabecera de C estándar. La línea:

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

nos indica que la variable `dht` será de clase `DHT` y se configura con el pin de datos y con el tipo de sensor, en nuestro caso un DHT21; el resto de opciones para este parámetro son DHT11, DHT21 y AM2301, todas ellas definidas en el fichero de cabecera `DHT.h`. Por último hay que comentar que se realizan las lecturas y luego para ver si hay error se usa la función `isnan()`; esta función devuelve 1 si el parámetro que se le ha pasado no es un número y 0 en caso contrario, *isnan* significa *is not a number* (no es un numero), por lo que sirve para conocer si un valor dado corresponde a un número o no; en este caso, si las lecturas fallaran la propia función de la librería devuelve la constante `NAN` (*Not A Number*, no es un número).
Ahora para mejorar este circuito invito al lector a modificarlo para que en vez de mostrar la salida en el monitor serie, la muestre en un display de 7 segmentos o en la pantalla de algún programa de ordenador.

Joystick

El joystick es un dispositivo que dispone de una pieza central que pivota sobre su base y permite informar su ángulo de inclinación o desplazamiento lineal y dirección al elemento que controla, aunque hay joysticks que pueden informar uno y tres ejes lo normal es que den información sobre dos ejes; más concretamente lo normal es que se usen para obtener datos referentes al plano (X,Y); desplazando el joystick hacia la izquierda y derecha para moverse por el eje X y arriba y abajo para el eje Y. Se trata de un dispositivo analógico que dependiendo de lo que se incline la palanca hacia cada lado, presentará unos valores u otros en sus terminales.
Lo normal es que en estos dispositivos en reposo se dé una lectura neutra, indicando que ningún eje debe modificarse, es decir una lectura 0 de variación. Como se trata de un dispositivo analógico lo que se tiene es una lectura de la mitad de 1023 en caso de usar 10 bits de conversión, correspondiendo

valores mayores a 511 si se desplaza hacia un lado y valores menores si se desplaza hacia el contrario. Cómo es difícil que los dos ejes presenten la misma calibración, algunos modelos de joysticks analógicos incorporan unos pequeños potenciómetros para poder calibrar cada eje independientemente y obtener así un estado de reposo idéntico en ambos ejes y que muestren lecturas de 511. Si disponemos de un joystick que no tenga esta opción, siempre podemos calibrarlos añadiendo un divisor de tensión a la salida con un potenciómetro externo.

Aunque son más comunes los joysticks analógicos, también existen joysticks digitales, donde simplemente se obtiene en su lectura 0 o 1 dependiendo de si está desplazado o no.

Normalmente, además de la palanca de control sobre los ejes, los joysticks suelen incorporar botones digitales adicionales, por lo que en cuanto a terminales del dispositivo se tendrán el de alimentación, el de tierra, uno de lectura de la posición de cada eje y terminales adicionales para la lectura de cara uno de los botones que incorpore.

Los botones incorporados en ocasiones no son claramente visibles, así por ejemplo en la figura 8.5 podemos ver un joystick con capucha y sin ella; el botón se encuentra entre la palanca de control y las patillas.

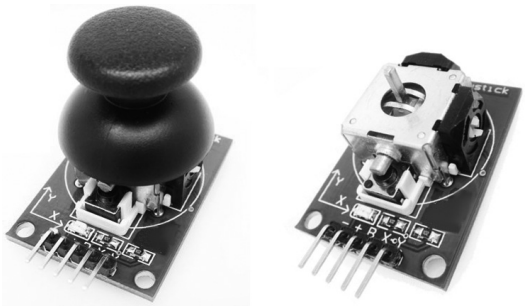


Figura 8.5. Joystick analógico.

La manera de cablear cada dispositivo es distinta dependiendo del fabricante, por lo que se debe poner atención a la serigrafía del componente ya que pueden tener las patillas en diferente orden.

Para comprender mejor su funcionamiento realizaremos un pequeño ejemplo donde controlaremos una nave en el ordenador mediante el joystick.

Las conexiones como hemos dicho son muy sencillas, simplemente se debe alimentar el sensor con 5V y conectar su tierra; por parte de los ejes la lectura se realizará utilizando dos entradas analógicas y para el estado del botón usaremos una entrada digital.

Para el circuito no necesitaremos nada adicional, simplemente el joystick convenientemente cableado.

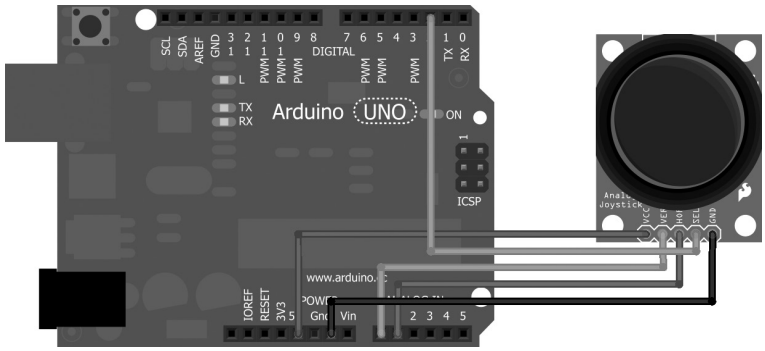


Figura 8.6. Circuito con joystick analógico.

Para el *sketch* podemos comenzar por mostrar los valores leídos en el terminal serie donde se observarán los valores que van tomando cada una de las lecturas dependiendo de cómo movamos el joystick o apretemos el pulsador.

```
const byte pinX = A0;
const byte pinY = A1;
const byte pinButton = 2;
void setup(){
   Serial.begin(9600);
   pinMode(pinButton,INPUT);
   digitalWrite(pinButton,HIGH);
   pinMode(pinX,INPUT);
   pinMode(pinY,INPUT);
}

void loop (){
   int coordX = analogRead(pinX);
   int coordY = analogRead(pinY);
   boolean buttonState = digitalRead(pinButton);
   Serial.print("Coordenada X: ");
   Serial.print (coordX );
   Serial.print("  Coordenada Y: ");
   Serial.print (coordY );
   Serial.print("  Pulsador: ");
   Serial.println (buttonState);
}
```

Pero habíamos dicho que lo que haríamos sería controlar una nave del ordenador, así que vamos a ponernos con ello. Lo que se hará será transmitir al ordenador los valores de las lecturas de los ejes del joystick y el estado del pulsador; dependiendo del valor de cada uno de los ejes la nave se desplazará más o menos rápido por la pantalla y en la dirección y sentido indicado

por el joystick; en caso de que se pulse el botón, cambiaremos el gráfico de la nave de modo que parezca que encenderemos los motores. Para la transmisión hacia el ordenador de las órdenes pertinentes desde el joystick necesitaríamos 3 bytes, uno para el valor del eje X, otro para el valor del eje Y y otro para el botón. Los valores de los ejes los transformaremos en valores de 0 a 100 en lugar de trabajar de 0 a 1023, esto quiere decir que en reposo, los valores de las lecturas de los ejes deberían ser 50.
Usaremos una transmisión serie para comunicarnos con el ordenador, pero para mejorar la transmisión y saber que dato es cada uno, utilizaremos un código de inicio de transmisión (una especie de *handshake*, sólo que en este ejemplo sólo hay un interlocutor, por lo que no es un *handshake* propiamente dicho pero sirve para nuestro cometido), enviaremos un primer byte con el valor 250 y esto indicará al ordenador que los tres siguientes bytes corresponden al valor de la coordenada X, al valor de la coordenada Y y al estado del botón; en este orden. Así lo que se hará en el *sketch* es realizar las lecturas de cada uno de los terminales, mapear los valores para cambiar a la escala 0-100 guardar los valores en un array de cuatro elementos cuyo primer byte es 250 (inicio de datos) y enviarlo por serie hacia el ordenador.

```
const byte pinX = A0;
const byte pinY = A1;
const byte pinButton = 2;
byte message[4]; // valores de las lecturas
void setup(){
   Serial.begin(9600);
   pinMode(pinButton,INPUT);
   digitalWrite(pinButton,HIGH);
   pinMode(pinX,INPUT);
   pinMode(pinY,INPUT);
}

void loop (){
   int coordX = analogRead(pinX);
   int coordY = analogRead(pinY);
   boolean buttonState = digitalRead(pinButton);
   message[0] =  250; // inicio de datos
   message[1] =  map(coordX,0,1023,0,100) ; // coord X escala de 0 a 100
   message[2] =  map(coordY,0,1023,0,100) ; // coord Y escala de 0 a 100
   message[3] =  buttonState; // pulsador
   Serial.write(message,4); // envío al ordenador
}
```

Para la parte del ordenador volveremos a usar Processing, pero podríamos utilizar Java, .Net o nuestro lenguaje de programación favorito con tal de leer los datos transmitidos; sólo con la premisa de que se envían secuencias de bytes donde el primero de ellos llegará con valor 250 que marca el inicio de los datos a procesar. El listado completo para Processing es el siguiente:

```
import processing.serial.*;

Serial myPort;  // Create object from Serial class
PImage img0; // imagen nave
PImage img1; // imagen nave y motor
PImage bg; // fondo
boolean isNormal = true; // control del estadode l motor
// posiciones y velocidades de la nave
float x = 0;
float y = 100;
float speedX = 0;
float speedY = 0;
// dimensiones de la nave
int RECT_W = 158;
int RECT_H = 42;

void setup() {
   size(400, 400); // tamaño de pantalla
   println(Serial.list());
   myPort = new Serial(this, Serial.list()[13], 9600);
   // carga de imágenes
   img0 = loadImage("ship0.png");
   img1 = loadImage("ship1.png");
   bg = loadImage("bg.png");
}

void draw() {
   background(255);
   move();
   display();
   while (myPort.available () > 0) {
      byte[] inBuffer = new byte[3];
      // desechamos las lecturas hasta encontrar un 250 que marca el incio
      myPort.readBytesUntil(250);
      // los tres siguientes bytes son de datos, recoger los 3
      if (myPort.readBytes(inBuffer)==3) {
         // las velocidades las mapeamos de -2 a 2, para poder tener
         // aceleraciones negativas
         speedX = map(inBuffer[0], 0, 100, -2, 2);
         speedY = map(inBuffer[1], 0, 100, -2, 2);
         // detección de pulsación
         if (inBuffer[2]== 1) {
            isNormal = true;
         }
         else {
            isNormal = false;
         }
      }
   }
}

/*******************************************************
* Actualiza la posición de la nave teniendo en cuenta
* las velocidades de los ejes y los bordes
* de la pantalla para no salirse de ella
*******************************************************/
```

```
void move() {
   // eje X
   x = x + speedX;
   if (x > (width - RECT_W)) {
      x = width - RECT_W;
   }
   else
   if (x<0) {
      x=0;
   }
   // eje Y
   y = y + speedY;
   println(y);
   if (y > (height - RECT_H)) {
      y = height - RECT_H;
   }
   else
   if (y<0) {
      y=0;
   }
}

// Dibujamos la pantalla
void display() {
   image(bg, 0, 0, 400, 400);
   if (isNormal) {
      image(img0, x, y, RECT_W, RECT_H);
   }
   else {
      image(img1, x, y, RECT_W, RECT_H);
   }
}
```

Lo primero que hemos hecho ha sido definir una serie de variables que nos ayudarán a mantener datos de la aplicación como las imágenes de la nave o la posición y velocidad de ésta. Durante la configuración realizada en la función `setup()`; definimos el tamaño de la pantalla, cargamos en memoria las imágenes y configuramos la conexión serie, tal y como se hizo en el ejemplo de transmisión serie. En este punto hay que ajustar el valor del puerto al que se encuentre conectada la tarjeta Arduino. Durante la función `draw()`, que es como la función `loop()` de Arduino, nos encargamos de calcular las nuevas coordenadas de la nave con los parámetros que se tengan en ese momento (mediante la función `move()`), dibujar esa nueva posición (usando la función `display()`) y recuperar una nueva lectura de datos.

En la lectura de datos se mira a ver si hay bytes esperando en el buffer de entrada, en tal caso se desechan entonces todos aquellos que se encuentren hasta encontrar un byte con el valor 250 que indica el comienzo de la transmisión mediante:

```
myPort.readBytesUntil(250);
```

Una vez se ha leído el byte con 250, los tres siguientes corresponden a los valores de los ejes del joystick y al pulsador, por lo que se leen 3 bytes y se comprueba que realmente se han leído 3 (si sólo hubiera dos bytes es que no ha dado tiempo a la transmisión y se desechan también los datos.

```
if (myPort.readBytes(inBuffer)==3) {
   // procesar datos
}
```

En nuestro caso este mecanismo es válido porque no es importante perder lecturas, ya que se envían continuamente y el no recibir alguna lectura no afecta al funcionamiento; en caso de ser importante cada dato transmitido habría entonces que implementar un *handshake* completo y unos códigos de redundancia o bytes de *checksum* para asegurarnos de que los datos son correctamente recibidos. Los datos recibidos para los ejes están en la escala de 0 a 100, por lo que no nos valen para mover la nave en sentido negativo en los ejes, así que se realiza una traslación de valores a la escala -2 a 2 y se almacena en las variables para guardar las velocidades.

```
speedX = map(inBuffer[0], 0, 100, -2, 2);
speedY = map(inBuffer[1], 0, 100, -2, 2);
```

Mediante la función `move()` calculamos la nueva posición de la nave, siempre teniendo en cuenta no salirse de los márgenes de la pantalla, por eso por ejemplo si la coordenada `x` es mayor que el ancho de la pantalla menos el tamaño de la nave quiere decir que comenzaría a salirse la nave por la derecha, así pues hay que reajustar su coordenada para que se quede pegada al borde de la ventana.

```
x = x + speedX;
if (x > (width - RECT_W)) {
   x = width - RECT_W;
}
```

Lo mismo hacemos en caso de que sea menor que cero, que en este caso se saldría por el borde izquierdo de la pantalla. Estas consideraciones se deben tener también con el eje Y para evitar que perdamos la nave por el borde superior o inferior de la ventana.
Por último la función `display()` nos muestra el fondo de pantalla y luego dependiendo del valor leído en el byte 3 (que corresponde al estado del pulsador) y que se ha guardado en la variable `isNormal`, mostrará la imagen `img0` o la `img1`, que se corresponden a la nave en estado normal o con los motores encendidos.

```
if (isNormal) {
   image(img0, x, y, RECT_W, RECT_H);
}
```

```
else {
   image(img1, x, y, RECT_W, RECT_H);
}
```

La función `image()` toma como parámetros la imagen a mostrar previamente cargada, las coordenadas de origen y el ancho y alto de la imagen a mostrar que no tienen por qué corresponder con el tamaño de la imagen cargada (es decir que la podemos distorsionar). El aspecto del programa en ejecución es el mostrado en la figura 8.7.

Figura 8.7. Programa controlado con joystick en funcionamiento.

Con un poco más de código ya podríamos hacer nuestro propio juego controlado por Arduino... de hecho le animo a intentar hacer el juego Pong para dos jugadores controlados por joystick o por acelerómetro.

9 Infrarrojos

En este capítulo aprenderá a:

- Descubrir las características luminosas de la luz infrarroja.
- Cómo funcionan los emisores infrarrojos.
- Recibir y entender los datos transmitidos por un emisor.
- Conocer utilidades de la luz infrarroja.
- Utilizar un detector de presencia infrarrojo.

La radiación infrarroja fue descubierta por el astrónomo Sir Frederick William Herschel en 1800; Herschel estaba interesado en descubrir cuánto calor atravesaba unos filtros coloreados que utilizaba para observar el sol, ya que había notado que dependiendo del color se calentaban más o menos. Para medir la temperatura hizo pasar la luz solar por un prisma de cristal, descomponiendo la luz y obteniendo el arco iris; sobre estos colores puso distintos termómetros y notó que según avanzaba por los colores violeta, azul, verde, amarillo, naranja y rojo la temperatura medida por los termómetros iba incrementando. La sorpresa fue cuando colocó un termómetro más allá del color rojo, donde obtuvo la mayor de las temperaturas, descubriendo así que existía una luz o radiación no visible y que denominó rayos caloríficos. Estos rayos caloríficos eran lo que actualmente conocemos como rayos infrarrojos o simplemente infrarrojos.

La luz infrarroja (IR) es principalmente radiación térmica y es que todos los objetos que tengan una temperatura superior a los 0K o -273.15°C (el cero absoluto) irradia energía en la banda infrarroja; esto es la base de las cámaras de visión nocturna y la termofotografía. De modo más cotidiano, la luz infrarroja nos la podemos encontrar en los lectores de códigos de barras, detectores de puertas de garaje, toma de medidas a distancia o en el propio mando de la televisión.

Si hablamos más en el aspecto físico, la luz infrarroja se emite a una frecuencia de entre 430THz y 300GHz y con una longitud de onda entre 700nm y 1mm, es decir fuera del alcance de la visión humana; dentro de la luz infrarroja, se clasifica en el infrarrojo cercano que comienza sobre los 700nm y el infrarrojo lejano que va de los 100µm a 1 mm.

Normalmente los emisores de luz infrarroja se ven acompañados de un receptor para darle utilidad. Dado que la radiación infrarroja no deja de ser una emisión de luz, es muy fácil implementar en los circuitos un led que emita este tipo de radiación lumínica para poder así transmitir información (distancias cortas) hacia un receptor y dado que está fuera del alcance de la visión humana realizará la transmisión de modo no intrusivo, totalmente transparente para el usuario.

Receptor infrarrojo

El receptor infrarrojo es una fotorresistencia como la que vimos anteriormente pero con ciertas variaciones. En las fotorresistencias ya vistas, se tenían dos terminales y dependiendo de la cantidad de luz variaba su resistencia, de modo que mediante un circuito de divisor de tensión obteníamos resultados

es decir se comportaba de modo analógico; además normalmente trabajan en el espectro visible de luz. En el caso del receptor infrarrojo, podemos encontrarlo mayormente de dos formas, encapsulado en un elemento de dos patillas (que habrá que habrá que montar con un divisor de tensión para su lectura) o en unos elementos electrónicos de tres terminales que tendremos que alimentarlos individualmente y donde existe una patilla donde se obtendrán, dependiendo de lo detectado por el sensor, 5V ó 0V en caso de ser detectores digitales, o un valor de 0V a 5V en caso de los analógicos.
La detección de la luz se realiza mediante demodulación de la luz recibida, es decir su funcionamiento no es mediante luz constante, sino mediante pulsos emitidos que se deben interpretar. La mayor parte de los receptores infrarrojos domésticos incorporan un demodulador que trabaja a 37.5-40KHz y que será el encargado de indicar cuándo se debe mostrar salida `HIGH` en el terminal correspondiente del receptor; cuando detecte emisiones infrarrojas en su frecuencia (por ejemplo a 38KHz), la salida obtenida será de 5V mientras que cuando no se detecte será de 0V.

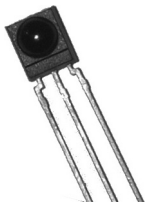


Figura 9.1. Receptor infrarrojo.

Tanto los emisores como los receptores trabajan en pareja; dado que la luz infrarroja puede emitirse en distintas longitudes de onda, deberemos escoger un receptor que sea excitable a la longitud de onda del emisor; o dicho de otra manera, sabemos que el emisor va a transmitir señales que se emitirán a una cierta frecuencia, pues bien, el receptor debe trabajar de igual modo a esta frecuencia con tal de poder interpretar correctamente los datos transmitidos por el emisor.
Para ver cómo actúa el receptor vamos hacer un primer circuito de modo que la patilla de señal alimente un led de espectro visible (un led de los que hemos estado usando hasta ahora), de este modo cada vez que el receptor reciba algo, se encenderá el led. Para el circuito simplemente necesitaremos un led rojo (por ejemplo), una resistencia de 220Ω, un receptor de rayos infrarrojos y un emisor de infrarrojos (que puede ser el mando a distancia de la televisión). Cómo receptor de infrarrojos se puede utilizar cualquiera de los que hay en el mercado, procurando que la recepción la efectúe en el rango

de los 38KHz ya que es una frecuencia muy común en los emisores domésticos, un ejemplo de receptor de este tipo es el PNA4602M de Panasonic o la familia TSOP312XX donde XX es la frecuencia a la que trabajan, en nuestro caso sería el TSOP31238. Para este circuito no necesitaremos ningún *sketch*, simplemente se tomará la emisión del mando a distancia como entrada del detector y éste encenderá y apagará el led dependiendo de la salida proporcionada en su terminal. El montaje es el correspondiente la figura 9.2.

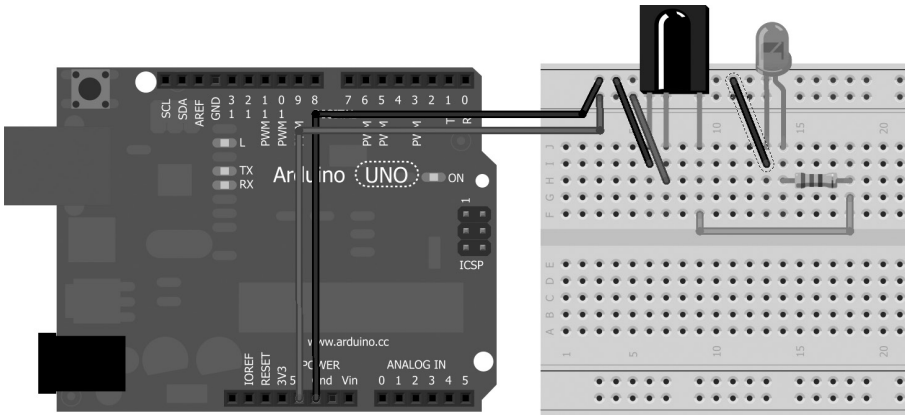


Figura 9.2. Circuito de prueba de receptor infrarrojo.

Si apuntamos con un control remoto al receptor, veremos cómo se va iluminando el led a intervalos; cada vez que se ilumina se está transmitiendo algún dato por parte del emisor (el mando de la televisión).
Si no somos capaces de ver nada en el led, puede ser que esté mal montado el circuito, que esté el led fundido, que no funcione el receptor o que el emisor no esté emitiendo. Para revisar el circuito no hay que explicar nada, para comprobar si el led está fundido, podemos alimentar su ánodo directamente con 5V y se iluminará, pero para probar el emisor tenemos un problema, ya que la luz que emite no es visible al ojo humano. Efectivamente no es visible para el ojo humano pero si para las cámaras digitales, así que para probar si funciona vale con poner la cámara de nuestro móvil, apuntar con el mando hacia la lente y mientras se aprieta algún botón del mando ir observando la pantalla del móvil y en caso de que funcione correctamente, en la pre visualización de la fotografía podremos ver una lucecita que se enciende y apaga a ráfagas.
Adelantándonos un poco a un ejemplo de este libro, si se quisiera hacer un detector de presencia, lo que se podría hacer es enviar una ráfaga cada 5ms y en el caso de no recibir nada durante 10ms significaría que hay algo

entorpeciendo el camino de la luz y por tanto detectaríamos que hay presencia. Esta será la manera de actuar de un detector de presencia (más adelante lo veremos con más profundidad) pero este no es el caso que nos interesa ahora, sino que nos interesa el del uso de la luz infrarroja como control. El mando a distancia de la televisión envía unas ráfagas de luz infrarroja que la televisión interpreta y sabe si debe cambiar de canal o modificar su volumen... Pero ¿cómo lo hace?

Las ráfagas que vemos en la pantalla de la cámara fotográfica, corresponden con la información recibida; tal y como se ha dicho, el receptor muestra en su terminal de datos 5V o 0V dependiendo de si recibe o no excitación por parte de la radiación infrarroja, así que trabajaremos de modo semejante a como lo hicimos con el sensor de temperatura, detectando cuánto tiempo está el receptor en 5V y cuanto tiempo está en 0V y dependiendo de las combinaciones de tiempos tendremos la codificación del "comando" enviado por el control remoto.

Para detectar la codificación enviada realizaremos un pequeño *sketch* capaz de detectar los tiempos de recepción de señal y poder así conocer el dato enviado; utilizaremos como fuente emisora un mando a distancia y capturaremos la señal, obteniendo una lista de los tiempos usados para cada comando emitido hacia nuestro circuito.

El circuito utilizado para poder capturar las señales emitidas será muy semejante al del ejemplo anterior, solo que en este caso no existe en led de aviso de señal y la señal se envía directamente al terminal 2 de la tarjeta Arduino; será entonces de ese terminal de donde nuestro *sketch* tenga que leer y procesar los datos.

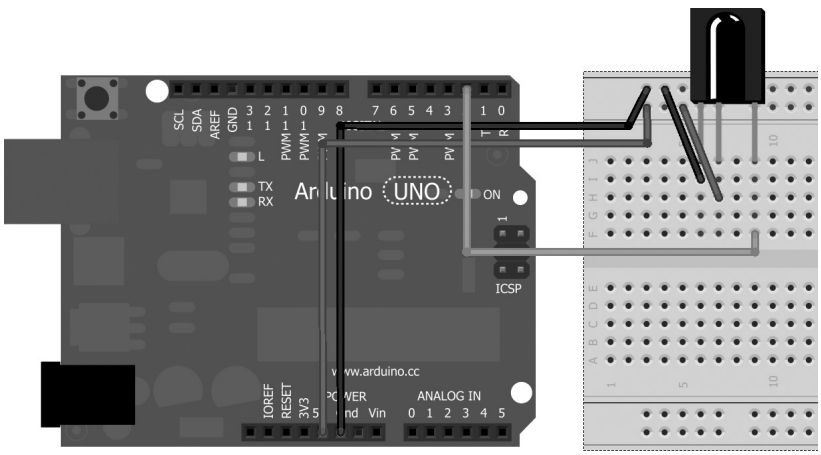


Figura 9.3. Circuito de captura de receptor infrarrojo.

Lo que debemos realizar a groso modo en este *sketch* es ir leyendo de la entrada de la tarjeta Arduino, teniendo en cuenta cuánto tiempo está la señal en `HIGH` y cuanto está en `LOW` e irlo guardando para luego, más adelante en otro ejemplo, poder repetir la ráfaga y simular el dato enviado por el mando a distancia. El problema que vamos a tener en este *sketch* es que los tiempos de las ráfagas de emisión son muy cortos y la función de lectura habitual para Arduino (la función `digitalRead()`) es más lenta de lo deseado, por lo que habrá que realizar algún truco para obtener lecturas más rápidas.
El *sketch* lo comenzaremos añadiendo una serie de variables que nos ayudarán en la tarea de lectura de la señal, como por ejemplo la definición del pin que usaremos para la lectura. Necesitaremos también alguna manera de indicar el tiempo máximo de pulso, es decir aquel tiempo que superado podamos decir que eso ya no es un pulso, sino que no se emite nada, para ello creamos la constante `MAXPULSE`. Por otro lado para controlar la granularidad de la escucha definiremos otra constante llamada `RESOLUTION`, que se usará como tiempo de espera durante el pulso para no estar atendiendo todo el rato a la entrada. Por último necesitaremos algún lugar donde guardar las señales recibidas, para ello crearemos un array de 100 elementos que cada uno de ellos guardará el tiempo que ha estado la señal en `HIGH` y en `LOW`, por lo que será un array de dos dimensiones; acompañando al array tendremos un índice que nos servirá para saber en qué posición del array nos encontramos en cada momento.

```
// pin de escucha
const byte irPin = 2; // pin de escucha
// máximo pulso a escuchar
const unsigned int MAXPULSE = 50000;
// resolución de escucha
const unsigned int RESOLUTION = 10;
// parejas de envío
unsigned int pulses[100][2]; // la pareja representa el HIGH y el LOW
// indice para recorrer los pulsos
byte pulseIndex = 0;
```

En la función de `setup()` simplemente nos debemos de preocupar de preparar la conexión con el monitor serie donde iremos mostrando los datos obtenidos en las lecturas.
Dentro de la función `loop()` deberemos ir leyendo el terminal 2 e ir guardando los tiempos en los que está en `HIGH` y los que está en `LOW`; el problema es que estamos hablando de tiempos de microsegundos y las señales captadas deben ser lo más próximas posibles a lo enviado con tal de poder ser reproducidas más adelante sin problemas, con lo que la lectura sobre el pin 2 que haríamos normalmente:

```
while (digitalRead(irPin)) {
```

no nos sirve, así que usaremos otro método más rápido que será leer directamente de los registros de los puertos del microcontrolador. Los registros de los puertos permiten trabajar a bajo nivel con los datos disponibles en los pines de la placa Arduino, y realizar lecturas y escrituras más rápidas que con las funciones clásicas como `digitalRead()`.
Los microcontroladores ATmega8 y ATmega168 poseen 3 puertos:

- Puerto B que incluye los pines digitales del 8 al 13.
- Puerto C que incluye los pines analógicos.
- Puerto D que incluye los pines digitales del 0 a 7.

Cada uno de los puertos está controlado por tres registros que son también variables dentro del lenguaje Arduino y son `DDR`, `PORT` y `PIN`. El registro `DDR` indica si el pin es de tipo `INPUT` o `OUTPUT`; el registro `PORT` controla si está en `HIGH` o `LOW` y el registro `PIN` lee el estado de los pines que se hayan configurado de tipo `INPUT` mediante la instrucción `pinmode()`. Tanto `DDR` como `PORT` son de lectura y escritura, mientras que el registro `PIN` es sólo de lectura.
Por lo tanto nos interesa leer el registro `PIN` del puerto D, dado que estamos intentando leer el valor del terminal digital 2 de la tarjeta. El registro `PIN` tiene 8 bits y para realizar la lectura del terminal o terminales que nos interesen debemos colocar a 1 el número de bit correspondiente al terminal a leer; es decir, para leer el terminal digital 2 dentro de este registro, debemos activar el bit en índice 2, para ello utilizaremos una función que retorna un byte devolviendo a 1 el número de byte indicado como parámetro, se trata de `_BV(numbyte)`; si es invocada mediante `_BV(2)` nos devolverá el byte de índice 2 (es decir el que se encuentra en tercera posición desde la derecha, recuerde que empieza a contar en 0) con valor 1 y el resto a 0; así pues juntando lo visto en este párrafo, para realizar una lectura rápida del estado del pin haremos un *and* del registro y 00000100 para quedarnos con el valor de la entrada 2. Usaremos el código:

```
while (PIND & _BV(irPin)) {
```

Dentro de este `while()` permaneceremos mientras la entrada leída sea `HIGH`, por lo que deberemos de ir guardando este tiempo; en este caso por comodidad lo haremos en una variable local que luego guardaremos en el array de tiempos totales. En lugar de leer cada microsegundo la entrada, para no cargar al microcontrolador, lo que se hace es leer cada `RESOLUTION` tiempo, realizando un `delay()` y que luego más adelante ya ajustaremos, ya que si en cada ciclo aumentamos en 1 el tiempo que está en `HIGH`, realmente no aumentamos en 1, sino en `1*RESOLUTION`.

```
// contamos microsegundos
highPulse++;
delayMicroseconds(RESOLUTION);
```

Del bucle saldremos tan pronto la entrada leída sea negativa o si llevamos esperando en `HIGH` más tiempo de lo marcado por la constante `MAXPULSE`; es una manera de marcar un *deadline* y que no se quede colgada la aplicación, ya que si no se reciben pulsos puede ser que no haya nada a la entrada o que se haya terminado de emitir el comando, que en tal caso vendría dado por que el índice `pulseIndex` que marca la posición dentro del array de señales recibidas sea distinto de 0; en este caso mostramos por pantalla el resultado e iniciamos de nuevo el índice.

```
if ((highPulse >= MAXPULSE) && (pulseIndex != 0)) {
   // Imprimimos en pantalla lo obtenido hasta el momento
   printPulses();
   pulseIndex=0;
   return;
}
```

Al acabar el bucle guardaremos el valor del tiempo en estado `HIGH` obtenido, dentro del array `pulses`, en el índice marcado por `pulseIndex` que guarda un array de dos posiciones; dentro de este array, guardaremos el estado `HIGH` en la posición 0 (dejando la posición 1 para los `LOW`).

```
pulses[pulseIndex][0] = highPulse;
```

Del mismo modo se trabajará con el estado `LOW`; se debe realizar un bucle que atienda la entrada mientras ésta se encuentra en estado `LOW` e ir guardando el tiempo en el que se está en este estado; al finalizar el bucle bien por tener un pulso mayor que `MAXPULSE` o por cambiar de estado la entrada, se debe guardar el tiempo obtenido en el array `pulses`, esta vez en la posición 1 del array.

```
pulses[pulseIndex][1] = lowPulse;
```

También como se hizo en el caso del estado `HIGH`, si se produce un pulso mayor que `MAXPULSE` y el índice `pulseIndex` es distinto de 0, se debe mostrar por pantalla lo obtenido hasta el momento mediante la función `printPulses()`.
Tras acabar el ciclo de lectura en `LOW`, se debe actualizar también el índice que recorre el array `pulses` para apuntar a la siguiente posición.

```
pulseIndex++;
```

En la función `printPulses()` mostraremos lo almacenado en el array `pulses` y lo haremos de dos formas, una de lectura más humana y otra pensando en el siguiente ejercicio, que nos ahorrará mucho trabajo tedioso.

La forma más humana sería mostrando por filas el tiempo que ha estado en `LOW` y `HIGH` en microsegundos:

```
Serial.println("\n\n\tSe ha recibido: \nLOW \tHIGH");
for (byte i = 0; i < pulseIndex; i++) {
   Serial.print(pulses[i][0] * RESOLUTION, DEC);
   Serial.print(" us, ");
   Serial.print(pulses[i][1] * RESOLUTION, DEC);
   Serial.println(" us");
}
```

Mientras que la forma menos humana mostrará en pantalla un array preparado para utilizar en otro *sketch* Arduino y que aprovecharemos en el siguiente ejemplo:

```
Serial.println("//array enviado HIGH, LOW (en microsegundos)");
Serial.println("unsigned int pulses[] = {");

for (byte i = 0; i < pulseIndex-1; i++) {
   Serial.print(pulses[i][1] * RESOLUTION , DEC);
   Serial.print(", ");
   Serial.print(pulses[i+1][0] * RESOLUTION , DEC);
   Serial.print(",");
}

Serial.print(pulses[pulseIndex-1][1] * RESOLUTION , DEC);
Serial.print(", 0};");
```

Juntando todas las piezas obtendríamos el *sketch* completo:

```
const byte irPin = 2; // pin de escucha
// máximo pulso a escuchar
const unsigned int MAXPULSE = 50000;
// resolución de escucha
const unsigned int RESOLUTION = 10;
// parejas de envío
unsigned int pulses[100][2]; // la pareja representa el HIGH y el LOW
byte pulseIndex = 0; // indice para recorrer los pulsos

void setup(void) {
   Serial.begin(9600);
   Serial.println("Esperando a recibir infrarrojos...");
}

void loop(void) {
   // variables temporales para el tiempo
   unsigned int highPulse, lowPulse;
   highPulse = lowPulse = 0; // las iniciamos

   // while (digitalRead(irPin)) {  // no nos sirve, necesitamos algo más
                                    // rápido

   // ciclo de lectura HIGH
   while (PIND & _BV(irPin)) { // esto sí es rápido
```

```
      // la lectura es HIGH
      // contamos microsegundos
      highPulse++;
      delayMicroseconds(RESOLUTION);

      // Si nos pasamos del tiempo definido como MAXPULSE
      // quiere decir que no hay más pulsos ni datos que obtener
      // se termina el ciclo
      if ((highPulse >= MAXPULSE) && (pulseIndex != 0)) {
         // Imprimimos en pantalla lo obtenido hasta el momento
         printPulses();
         pulseIndex=0;
         return;
      }
   }
   // guardamos el tiempo obtenido
   pulses[pulseIndex][0] = highPulse;

   // ciclo de lectura LOW
   while (! (PIND & _BV(irPin))) {
      // la lectura es HIGH

      // contamos microsegundos
      lowPulse++;
      delayMicroseconds(RESOLUTION);
      // Si nos pasamos del tiempo definido como MAXPULSE
      // quiere decir que no hay más pulsos ni datos que obtener
      // se termina el ciclo
      if ((lowPulse >= MAXPULSE) && (pulseIndex != 0)) {
         // Imprimimos en pantalla lo obtenido hasta el momento
         printPulses();
         pulseIndex=0;
         return;
      }
   }
   // guardamos el tiempo obtenido LOW
   pulses[pulseIndex][1] = lowPulse;

   // avanzamos el índice dentro del array para la siguiente lectura
   pulseIndex++;
}

/* Muestra por pantalla y formateado el array pulses[] */
void printPulses(void) {
   Serial.println("\n\n\tSe ha recibido: \nLOW \tHIGH");
   for (byte i = 0; i < pulseIndex; i++) {
      Serial.print(pulses[i][0] * RESOLUTION, DEC);
      Serial.print(" us, ");
      Serial.print(pulses[i][1] * RESOLUTION, DEC);
      Serial.println(" us");
   }

   // Lo imprimimos convenientemente en forma de array
   Serial.println("//array enviado HIGH, LOW (en microsegundos)");
   Serial.println("unsigned int pulses[] = {");
```

```
   for (byte i = 0; i < pulseIndex-1; i++) {

      Serial.print(pulses[i][1] * RESOLUTION , DEC);
      Serial.print(", ");
      Serial.print(pulses[i+1][0] * RESOLUTION , DEC);
      Serial.print(",");
   }

   Serial.print(pulses[pulseIndex-1][1] * RESOLUTION , DEC);
   Serial.print(", 0};");

}
```

Si ejecutamos el *sketch* y apuntamos un mando a distancia contra el receptor de infrarrojos a la vez que pulsamos alguno de sus botones, veremos aparecer la secuencia de envío en el monitor serie. Por ejemplo en mi caso he utilizado el mando a distancia de un monitor OKI y he pulsado al botón de encendido, el resultado obtenido ha sido:

```
Esperando a recibir infrarrojos...
Se ha recibido:
LOW   HIGH
60656 us, 950 us
770 us, 1800 us
780 us, 940 us
790 us, 940 us
780 us, 930 us
780 us, 940 us
780 us, 940 us
780 us, 940 us
780 us, 940 us
1640 us, 940 us
780 us, 1810 us
770 us, 940 us
49270 us, 220 us
38340 us, 940 us
780 us, 1800 us
780 us, 940 us
790 us, 930 us
790 us, 930 us
760 us, 960 us
750 us, 970 us
780 us, 940 us
780 us, 940 us
1640 us, 940 us
780 us, 1800 us
780 us, 940 us
// array enviado HIGH, LOW (en microsegundos)
unsigned int pulses[] = {
950, 770,1800, 780,940, 790,940, 780,930, 780,940, 780,940, 780,940,
780,940, 1640,940, 780,1810, 770,940, 49270,220, 38340,940, 780,1800,
780,940, 790,930, 790,930, 760,960, 750,970, 780,940, 780,940, 1640,940,
780,1800, 780,940, 0};
```

Si nos fijamos bien veremos que el primer pulso `LOW` es muy grande, de hecho no lo necesitamos como parte de la transmisión del mando, por eso lo eliminamos en el array mostrado al final y que usaremos más adelante. También podemos observar que los valores se repiten, en este caso dos veces; es decir, el comando se envía una vez y tras finalizar la transmisión de este comando se vuelve a enviar. Esto lo hacen muchos fabricantes para asegurarse que el comando llega correctamente, aunque pulsemos una sola vez el botón, el comando se envía dos veces (o más); en ocasiones la repetición se realiza de modo consecutivo, sin intervalos de pausa y en otras ocasiones, como en este caso, se envían señales indicando que se va a repetir el comando. La zona donde comienza la repetición es:

```
49270 us, 220 us
38340 us, 940 us
```

El número de datos, formato y tiempo de transmisión dependen de cada fabricante, pero hay que tener cuidado porque hay ciertos modelos de mandos a distancia que en lugar de tener un solo emisor tienen dos leds emisores, e incluso algunos emiten en distintas frecuencias; este tipo de mandos debemos evitarlos para trabajar con este ejemplo. Para saber si tiene uno o dos leds podemos volver a utilizar el truco de la cámara de fotos del móvil y prestar atención si al pulsar un botón del mando se ven una o más fuentes de luz provenientes del mando a distancia.
Como curiosidad... ¿Alguna vez se ha preguntado que hace la barra de la Wii encima del televisor? Apunte con la cámara del móvil y descubrirá su secreto.

Emisor infrarrojo

El emisor infrarrojo es un led parecido a los que se vieron anteriormente pero que en este caso y a diferencia de los vistos, emite radiación en el espectro de luz no visible. También se diferencia en que está diseñado para trabajar a pulsos; de la misma manera que los leds visuales (como el rojo que hemos usado en varios ejemplos) se fabrican de modo que pueden estar mucho tiempo encendidos (pensemos en el tiempo que se pasa encendido el led de *standby* de la televisión), los diodos led emisores de infrarrojos están preparados para trabajar a pulsos, entregando mayor energía en menores tiempos. Véase la figura 9.4.
El circuito que vamos a realizar en este ejemplo será muy semejante a los que usábamos en los leds de espectro visible, usando una resistencia limitadora de entre 180Ω y 220Ω. Véase la figura 9.5.

Figura 9.4. Emisor infrarrojo.

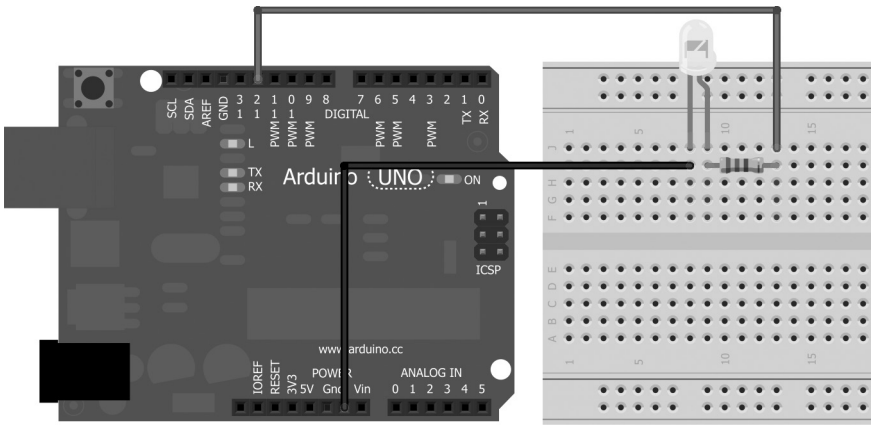


Figura 9.5. Circuito de emisor infrarrojo.

Lo que haremos en este *sketch* es reproducir lo captado por el *sketch* del ejemplo anterior, de modo que nuestro circuito simule ser el mando a distancia y si todo va de modo correcto, el dispositivo electrónico controlado por el mando a distancia debería actuar igual que si se pulsara la tecla del mando a distancia que se capturó.

Para trabajar en este ejemplo usaremos el pin 12 como terminal de envío de las señales, de modo que podemos dejar el circuito del receptor conectado a la tarjeta Arduino.

Como variables globales tendremos el pin sobre el que se enviarán las señales, el array capturado por el ejemplo anterior que debemos colocar en este caso como señales a enviar y una variable que mantendrá el tamaño de este array; dado que no sabemos el tamaño que tiene el array porque depende de la señal de cada fabricante, el tamaño lo calcularemos durante la función `setup()`.

```
const byte IRledPin =  12;
// array enviado HIGH, LOW (en microsegundos)
unsigned int pulses[] = {
950, 770, 1800, 780, 940, 790, 940, 780, 930, 780, 940, 780, 940, 780, 940,
780, 940, 1640, 940, 780, 1810, 770, 940, 49270, 220, 38340, 940, 780, 1800,
780, 940, 790, 930, 790, 930, 760, 960, 750, 970, 780, 940, 780, 940, 1640,
940, 780, 1800, 780, 940, 0};
byte arrSize = 0; // tamaño del array
```

La función `setup()` se debe encargar de configurar el pin de salida de la señal, el monitor serie y del cálculo del tamaño del array con las señales de infrarrojo para que luego sea más sencillo reproducirlas.

Nota:

Los pulsos a enviar por el emisor de infrarrojos, se deben ajustar a lo leído por el receptor durante el ejercicio anterior.

Para reproducir las señales se debe recorrer el array de pulsos teniendo en cuenta que los índices pares son salidas en `HIGH` y los impares son en `LOW`. Si el lector se siente más cómodo se puede variar el ejemplo anterior para que en lugar de generar en la salida un array de una sola dimensión nos genere un array de dos dimensiones con el `HIGH` y el `LOW` diferenciados.

```
for (int i =0; i < arrSize; ){
   pulseIR(pulses[i]);
   delayMicroseconds(pulses[i+1]);
   i +=2;
}
```

El envío de los pulsos se realiza teniendo en cuenta que la frecuencia de envío ha de ser de 38 KHz, que corresponde a estar 13 microsegundos en `HIGH` y 13 microsegundos en `LOW` y se tiene que enviar este tipo de pulsos durante el tiempo obtenido como que debe estar en `HIGH`, por lo que haremos una función que tenga como parámetro el tiempo a mostrar en `HIGH` y durante este tiempo enviaremos este tipo de pulsos.

```
void pulseIR(long microsecs) {

   cli();  // se desactivan las interrupciones

   while (microsecs > 0) {
      digitalWrite(IRledPin, HIGH);
      delayMicroseconds(10);
      digitalWrite(IRledPin, LOW);
      delayMicroseconds(10);

      // hemos tardado 26 microsegundos, ajustamos tiempo
      microsecs -= 26;
   }

   sei();  // se activan de nuevo las interrupciones
}
```

Puesto que es muy importante que la señal se envíe en unos tiempos determinados, esta función tiene un par de trucos para ajustarse lo máximo posible a la realidad. El primero de ellos es que desactivamos las interrupciones

del microcontrolador mediante la función `cli()` (se puede usar también la función `noInterrupts()`, que son idénticas), hacemos toda las tareas de envío y volvemos a activar las interrupciones mediante `sei()` (del mismo modo se puede usar también `interrupts()`); así, manteniendo las interrupciones desactivadas, nos aseguramos que durante el envío del pulso no se entorpece el programa por alguna interrupción del microcontrolador. El segundo truco viene del lado de los tiempos de escritura en el pin de salida, en el ejemplo anterior hemos visto como realizar las lecturas a través de los registros del puerto, en este caso, lo que vamos a hacer es en lugar de atacar a bajo nivel, ajustaremos los tiempos de emisión sabiendo que cada escritura `digitalWrite()` tarda aproximadamente 3 microsegundos, por eso se mantiene en `HIGH` y en `LOW` 10 microsegundos en lugar de 13 como debería ser porque 3 microsegundos los consume la propia escritura del estado.
Todo el *sketch* quedaría:

```
const byte IRledPin =  12;

// array enviado HIGH, LOW (en microsegundos)
unsigned int pulses[] = {
950, 770, 1800, 780, 940, 790, 940, 780, 930, 780, 940, 780, 940, 780, 940,
780, 940, 1640, 940, 780, 1810, 770, 940, 49270, 220, 38340, 940, 780, 1800,
780, 940, 790, 930, 790, 930, 760, 960, 750, 970, 780, 940, 780, 940, 1640,
940, 780, 1800, 780, 940, 0};

byte arrSize = 0; // tamaño del array

void setup() {
   // inicialización del pin de salida
   pinMode(IRledPin, OUTPUT);
   // cálculo del tamaño del array informado
   arrSize =  sizeof(pulses)/sizeof(int );
   Serial.begin(9600);
}

void loop(){
   Serial.println("Enviando infrarrojos");
   sendPulses();
   delay(30*1000);  // esperar 30 seg antes de volver a enviar
}

// Los pulsos son a 38 Khz, se deben enviar estos pulsos durante
// los microsegundos indicados por el parámetro de entrada
void pulseIR(long microsecs) {

   cli();  // se desactivan las interrupciones

   while (microsecs > 0) {
      // 38 kHz son  13 microsegundos en HIGH y 13 microsegundos en LOW
      // esta escritura tarda uno 3 microsegundos en producirse
      digitalWrite(IRledPin, HIGH);
```

```
      // se mantiene 10 microsegundos en HIGH, si no funciona se puede
      // probar con 11 o 9 microsegundos
      delayMicroseconds(10);
      // esta escritura tarda uno 3 microsegundos en producirse
      digitalWrite(IRledPin, LOW);
      // se mantiene 10 microsegundos en HIGH, si no funciona se puede
      // probar con 11 o 9 microsegundos
      delayMicroseconds(10);
      // hemos tardado 26 microsegundos, ajustamos tiempo
      microsecs -= 26;
   }

   sei();  // se activan de nuevo las interrupciones
}

// recorre el array de pulsos enviando uno a uno
void sendPulses() {
   for (int i =0; i < arrSize; ){
      pulseIR(pulses[i]);
      // esperamos en LOW el tiempo correspondiente
      delayMicroseconds(pulses[i+1]);
      // se ajusta el índice sumando dos, el HIGH y el LOW
      i +=2;
   }
}
```

Ahora podemos ejecutar el *sketch* apuntando el emisor de infrarrojos hacia el dispositivo electrónico controlado por el mando a distancia y el dispositivo electrónico reaccionará a lo enviado por el circuito. Para saber en qué momento se está enviando la señal, podemos abrir el monitor serie donde se informa de la transmisión o bien modificar el *sketch* para que se haga visible la transmisión a través del led del pin 13.

Detector de proximidad

Una de las aplicaciones de la emisión de infrarrojos es la detección de presencia, que consiste en que un led de infrarrojo está enviando continuamente pulsos y el receptor los va recibiendo; si el receptor deja de recibir los pulsos, quiere decir que hay algo que obstruye el paso de la luz del emisor hacia el receptor; se está detectando presencia. Este tipo de montajes se usan en muchos automatismos de líneas de producción, para saber cuándo ha llegado el producto que se debe procesar o también en puertas de garaje.
Otra de las aplicaciones de la luz infrarroja son los sensores de proximidad. Un sensor de proximidad permite conocer cuando un elemento se encuentra cerca de él, dependiendo del rango y de la calibración del sensor, se detectarán unas distancias u otras. El funcionamiento de los sensores de proximidad

en general es éste: el sensor posee un emisor de algún tipo de señal electromagnética o bien en forma de radiación visible o no visible (es el caso de los infrarrojos) y está de modo continuo emitiendo, a la vez que un receptor lee buscando cambios en las lecturas (bien del campo electromagnético o de la radiación, dependiendo del tipo de sensor) producidas por la proximidad de un elemento.
En el caso del sensor de proximidad de infrarrojos se compone de un led emisor y un receptor dispuestos de manera que el receptor recibirá la señal cuando rebote el algún objeto, de modo que si no hay nada delante el receptor no recibirá la señal y por lo tanto no detectará presencia.

Figura 9.6. Detector de proximidad infrarrojo.

Algunos de los detectores de proximidad por infrarrojos del mercado vienen ya preparados para utilizar directamente mientas que otros deberemos de poner resistencias limitadoras de tensión, del mismo modo algunos trabajan a 5V y otros a 3.3V por lo que se deberá tener cuidado a la hora de realizar la alimentación. Si el detector tiene algún potenciómetro (no todos lo tienen), éste nos servirá para calibrar la sensibilidad del receptor o incluso del emisor si tiene dos potenciómetros.

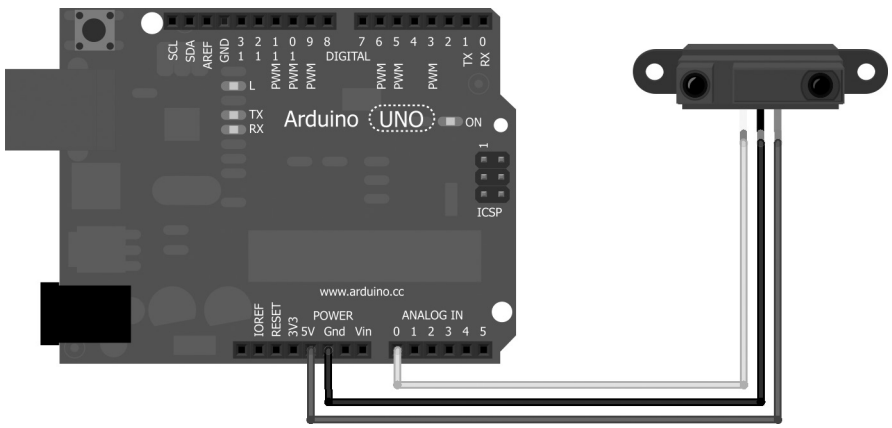


Figura 9.7. Circuito de detección de proximidad por infrarrojo.

En este ejemplo lo hemos conectado a una patilla analógica y es que podemos encontrar sensores de proximidad analógicos y digitales; en caso de ser analógicos el valor leído en la entrada irá aumentando a medida que la distancia con el objeto sea menor y en el caso de los digitales, se disparará el valor de salida del sensor de `LOW` a `HIGH` a un determinado umbral controlado por el potenciómetro de calibración o por las resistencias adicionales que se le añadan al circuito.

En el *sketch* de este ejemplo detectaremos la proximidad de un objeto mediante una lectura al terminal de entrada de la tarjeta Arduino cada medio segundo y en caso de detectar presencia se iluminará el led del terminal 13 (si el lector prefiere se puede conectar un altavoz piezoeléctrico a modo de alarma).

El *sketch* se tiene que preocupar de leer el terminal de entrada que en este caso es analógico y obtener su valor. Como cuanto mayor sea la proximidad del objeto mayor será el valor leído, tomamos como umbral 512 de modo que si la lectura es mayor que este valor encenderemos el led 13 y si es menor lo apagaremos.

```
int sensorPin = A0;
int ledPin = 13;
int sensorValue = 0;

void setup() {
   pinMode(ledPin, OUTPUT);
   Serial.begin(9600);
}

void loop() {
   // lectura del sensor
   sensorValue = analogRead(sensorPin);
   Serial.print("El valor obtenido es: ");
   Serial.println(sensorValue);
   // si es mayor de la mitad lo consideramos HIGH
   if (sensorValue > 512){
      digitalWrite(ledPin, HIGH);
   }
   else{
      digitalWrite(ledPin, LOW);
   }
   // detección cada medio segundo
   delay(500);
}
```

El *sketch* que se presenta está preparado para trabajar con lecturas analógicas, pero si se prefiere se puede modificar el circuito para que en lugar de leer en el terminal analógico 0 realice la lectura en alguno de los terminales digitales y luego en el *sketch* cambiar la lógica para que simplemente encienda el led 13 si la lectura del pin digital es `HIGH` en lugar de comprobar el valor leído con el umbral seleccionado anteriormente (valor 512).

10 Actuadores

En este capítulo aprenderá a:

- Conocer las opciones de motores más comunes.
- Utilizar un motor paso a paso.
- Usar servos eléctricos.
- Diferenciar un servo de un motor paso a paso.
- Controlar el flujo de corriente mediante relés.

En los últimos capítulos hemos ido viendo diferentes maneras de captar el entorno que rodea a la placa Arduino y hacerla partícipe de él mediante diferentes sensores; una vez que sabemos qué es lo que nos rodea, lo normal es hacer algo sobre el entorno, nosotros por ahora hemos ido mostrando información en una pantalla o hemos movido algo en un ordenador, pero siempre en el campo eléctrico/electrónico, vamos a ir un paso más allá, vamos a actuar sobre el entorno que nos rodea.
No existe una definición exacta de actuador, si bien podemos decir que son unos dispositivos que convierten una magnitud eléctrica en una salida generalmente mecánica, que puede provocar un efecto sobre el entorno que le rodea o sobre un proceso automatizado; en industria podemos encontrar actuadores neumáticos, hidráulicos o eléctricos. Como actuadores neumáticos podemos encontrar filtros, válvulas, pistones, reguladores... dentro de los hidráulicos tenemos acumuladores, bombas hidráulicas, pistones... En este capítulo veremos los más asequibles y fáciles de conseguir para comenzar nuestros prototipos que son los eléctricos, donde podemos encontrar entre otros los motores y relés.

Motores

Los motores son elementos capaces de transformar energía eléctrica en mecánica mediante campos electromagnéticos. Existen de muchos tipos entre los que podemos destacar:

- Motor de corriente continua: Funcionan con corriente continua con potencias que pueden ir desde unos pocos miliwatios a megawatios. Se les puede regular fácilmente la velocidad de giro y el par de fuerza aplicada. Son usados en elevadores, cintas de transporte, ventilación...
- Motor de corriente alterna asíncrono: Conocidos también como de jaula de ardilla debido a la forma característica de su rotor. Son motores muy baratos con potencias hasta varios kilowatios. Es posible regular su velocidad de giro, pero los elementos de la regulación son más caros que otras alternativas. Se usan en aplicaciones que requieran poca potencia y precisión como ventiladores o arranques de motores.
- Motor de corriente alterna de rotor bobinado: Trabaja con potencias de decenas de watios a varios kilowatios. Es posible regular la velocidad de giro mediante una precisión media. Podemos encontrarlo en aplicaciones de elevación, arranque de motores o como sustituto del motor de corriente continua.

- Motor paso a paso: Potencias y velocidades muy bajas. Permiten un posicionamiento excelente. Es ampliamente utilizado en la industria donde el posicionamiento de las piezas sea clave. Anteriormente podrían encontrarse en disqueteras de ordenador y en discos duros para mover los cabezales de lectura (no se anime a desmontar un disco duro actual ya que se dejó de usar hacia 1997; actualmente se usa un sistema llamado *voice coil* que permite un movimiento mucho más rápido de los cabezales) y hoy en día es ampliamente utilizado en otras áreas como el modelismo y la automatización de procesos.
- Servomotores: Trabaja con pequeñas potencias y velocidades de hasta 7000 rpm. Realmente es algo más que un motor ya que además incorpora una caja reductora de engranajes y un circuito de control. Son muy fiables a la hora de posicionarse y mantenerse en la posición seleccionada lo que les hace ideales para la industria. Mantienen muy estable el par de fuerza aplicada a distintas revoluciones.

A la hora de dar una solución a un problema mediante alguno de estos actuadores es muy probable que varios de ellos cumplan el objetivo buscado, en caso de de duda siempre se puede ir a un análisis de coste.

Motor	Compra	Instalación	Mantenimiento
C Continua	Alto	Medio	Alto
C Alterna asíncrono	Bajo	Bajo	Muy bajo
C Alterna bobinado	Alto	Medio	Alto
Paso a paso	Bajo	Bajo	Bajo
Servomotor	Medio	Medio	Bajo

Como podemos ver en la tabla, los que más sale a cuenta usar son los motores de corriente alterna asíncronos, los motores paso a paso y los servomotores. Puesto que no vamos a utilizar fuentes externas a la propia placa Arduino nos centraremos en los motores paso a paso y los servomotores.

Motores paso a paso

Dentro de los posibles modelos de motores a usar en electrónica, los motores paso a paso tienen un papel importante dado que permiten un control preciso del giro del motor a un precio asequible. También conocidos como *steppers*,

los motores paso a paso realizan su giro en pequeños pasos en lugar de tener un movimiento continuo (de ahí el nombre). Su rotor (la parte que gira del motor) tiene forma de rueda dentada mientras que su estator (la parte que no gira del motor) presenta una serie de electroimanes que se polarizan de forma intercalada (cuando uno de ellos está polarizado, los de sus lados no lo están), de modo que cuando se polariza uno de los electroimanes, atrae al diente que tiene más cercano y una vez que se tiene el diente frente al electroimán, se debe polarizar el electroimán siguiente de modo que atraiga de nuevo a ese diente y haga girar un poco al motor. Cada una de esas pequeñas rotaciones se denomina paso. Dependiendo de la "distancia entre dientes" del rotor, o dicho de otra manera dependiendo de la distancia de electroimanes del estator, tendremos más o menos pasos para completar una vuelta, cuanto menos espacio haya entre ellos, menores son los pasos a dar y más son los que se necesitan para completar una vuelta completa.

El número de pasos a dar para completar una vuelta completa del rotor depende de los pasos angulares del motor que están estandarizados (aunque pueden encontrarse algunos motores con pasos no estándares) y suelen ser de 1.8°, 5.625°, 7.5°, 11,25°, 18°, 45 y 90°. Por ejemplo para un paso angular de 1.8°, necesitaríamos 360°/1.8° = 200 pasos para completar la vuelta.

Los motores paso a paso más comunes son los motores unipolares, que presentan 5 o seis cables dependiendo de si tienen una masa común o bien diferentes masas por pares de bobinas y los motores bipolares que funcionan con cuatro cables.

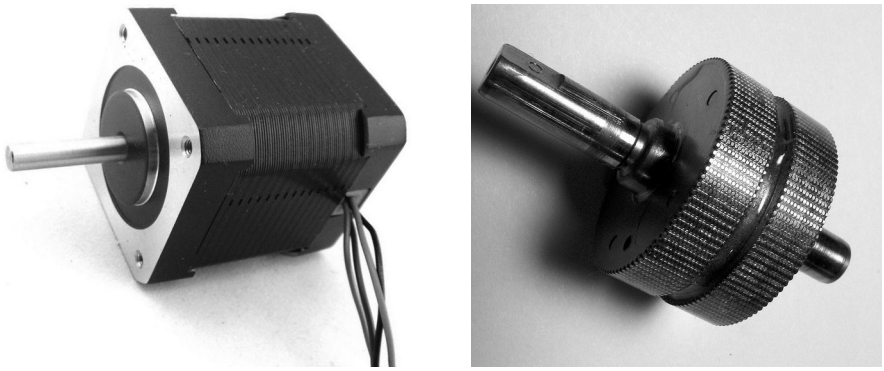


Figura 10.1. Motor paso a paso y rotor.

Dependiendo de si son bipolares o unipolares se deben realizar unas secuencias de polarización diferentes, teniendo en cuenta que los bipolares cambian el sentido del flujo de corriente por sus bobinas.

Para polarizar un bipolar la secuencia sería:

Paso	Terminal A	Terminal B	Terminal C	Terminal D
1	+V	-V	+V	-V
2	+V	-V	-V	+V
3	-V	+V	-V	+V
4	-V	+V	+V	-V

Mientras que para un unipolar la secuencia normal sería:

Paso	Bobina A	Bobina B	Bobina C	Bobina D
1	+V	+V	0	0
2	0	+V	+V	0
3	0	0	+V	+V
4	+V	0	0	+V

El funcionamiento básico es igual para ambos y es ir polarizando secuencialmente las bobinas para que vaya atrayendo el rotor hacia su posición y una vez que está en la posición adecuada, modificar la polarización del siguiente electroimán para desplazarlo de nuevo. Si no se desea desplazar, se mantiene la polarización del electroimán que se encuentre activo en ese momento, asegurando así un par que hace que no se mueva el rotor, es decir al estar polarizado el rotor no tiene movimiento libre, es como si presentara un freno. Esta también es una característica de los motores paso a paso, que en reposo tienen un consumo ya que se deben mantener polarizadas las bobinas para que no se desplace el rotor.

Existen algunos modelos de motores paso a paso que utilizan unos juegos de engranajes denominados reductores que consiguen que se puedan obtener muchos más pasos, por ejemplo podemos tener reducciones 1:64, que significa que por cada paso que da el motor, su movimiento se ve reducido 64 veces, así un motor de 200 pasos con reductora 1:64 realmente necesita 12800 pasos para dar una vuelta.

Normalmente los motores de paso a paso se utilizan junto con algún tipo de controlador que ayude a sincronizar las polarizaciones de los electroimanes con tal de conseguir el movimiento deseado a la velocidad deseada, sobre todo para los motores bipolares que son un poco más complicados de manejar ya

que es necesario cambiar la dirección del flujo de corriente por las bobinas. Esto no quiere decir que no podamos montarnos unos circuitos auxiliares para controlar nosotros mismos los motores, pero es mucho más sencillo utilizar un circuito integrado ya preparado que realizar, por ejemplo, dos montajes H-Bridge para controlar cada una de las bobinas de un motor bipolar.
Existen múltiples controladores y normalmente con el motor paso a paso el fabricante ya aconseja el uso de un controlador.
Para ver cómo funcionan los motores de manera práctica, haremos circuito en el que el motor paso a paso gire en sentido de las agujas del reloj y en sentido contrario de forma autónoma. Para el circuito necesitaremos un motor paso a paso y un controlador para el motor como pueden ser un L293, un ULN2003 o un ULQ2003.

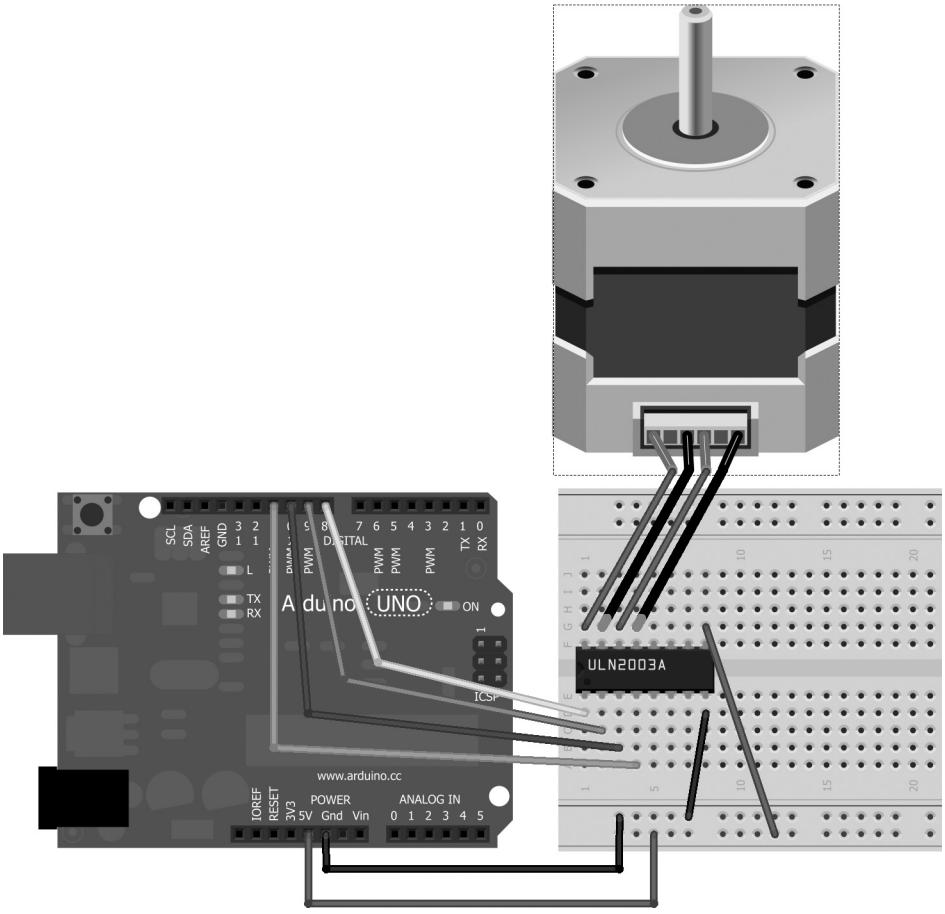


Figura 10.2. Circuito para motor paso a paso.

En caso de no conocer los pines de nuestro motor paso a paso por no venir con la serigrafía conveniente, mediante un multímetro en posición óhmetro es posible detectar las parejas de pines que forman cada bobina de excitación por existir continuidad eléctrica y poder así cablearlo correctamente.
Para trabajar con los motores paso a paso lo mejor es utilizar una librería que viene con el entorno de programación Arduino específicamente diseñada para este tipo de motores. Para poder utilizarla en los *sketchs* simplemente debe seleccionar el menú Sketch>Importar librería...>Stepper o incluir su archivo de cabecera mediante la línea:

```
#include <Stepper.h>
```

Este fichero permite trabajar con la clase `Stepper` y aislarnos de todas las complejidades a la hora de programar el motor. Lo primero que debemos hacer es generar una instancia de la clase `Stepper`, mediante la llamada:

```
Stepper(steps, pin1, pin2, pin3, pin4)
```

Donde `steps` son el número total de pasos por vuelta del motor que se va a utilizar. Si en lugar de los pasos, la característica que conocemos del motor es el paso angular (el ángulo que rota en cada paso), para obtener el número total de pasos por vuelta simplemente tenemos que dividir 360 por el paso angular.
Los parámetros `pin1`, `pin2`, `pin3` y `pin4` son los pines de Arduino a los que se ha conectado el motor. Existen controladores que sólo necesitan dos pines, en este caso el `pin3` y `pin4` no hace falta informarlos. Hay que tener especial cuidado en poner en orden los pines ya que de lo contrario no conseguiremos que el motor gire. Para indicar el número de pasos que se quieren ejecutar se utilizará la función `step(numPasos)` sobre la instancia de la clase `Stepper`, siendo `numPasos` el número de pasos que se quieren ejecutar; si el número de pasos es positivo girará en sentido de las agujas del reloj y si es negativo girará en sentido contrario. Por ejemplo si es un motor de 200 pasos para girar 90° haríamos:

```
stepper.step(50);
```

Para calcular lo que debe girar el motor paso a paso se debe tener en cuenta si dispone de engranajes reductores o no, dado que entonces el número de pasos a dar para completar la vuelta se ve aumentado en la ratio marcada por la reducción.
De una manera muy sencilla también podemos indicar la velocidad a la que queremos que se produzcan estos pasos. La velocidad se expresa en rpm y se realiza igualmente sobre la instancia de la clase por medio de la llamada `setSpeed(velocidad)`, donde `velocidad` son las revoluciones por

minuto a las que debe girar el motor. Una vez se le ha indicado la velocidad a la que debe girar, cuando se le indica que tiene que dar un número determinado de pasos, automáticamente se calcula cada cuanto se debe enviar una señal para generar el paso de modo que cumpla la velocidad indicada, nosotros no debemos preocuparnos de calcular los intervalos de señal.
Si generamos el *sketch* para que el motor gire en sentido horario y luego en sentido contrario con unas pausas de 2 segundos quedaría algo semejante a:

```
#include <Stepper.h>

const int STEPS = 100;   // Número de pasos por vuelta
/* Para conectar el motor necesitamos 4 pines
las entradas In1, In2, In3, In4  se ejecutarán
en secuencia  1-3-2-4  así lo indicamos
en la construcción del objeto */
Stepper stepper(STEPS, 8, 10, 9, 11);

const int stepsToPerform =1000; // pasos a ejecutar

void setup(){
   stepper.setSpeed(100);    // velocidad de rotación
}

void loop(){
   // Giro en sentido horario
   stepper.step(stepsToPerform);
   delay(2000);

   // giro en sentido anti horario
   stepper.step(-1*stepsToPerform);
   delay(2000);
}
```

En este ejemplo se ha configurado un motor de 100 pasos por vuelta (paso angular de 3.6°), con una velocidad de 100 revoluciones por minuto y va a ejecutar 100 pasos, es decir una vuelta entera. Si lo cargamos sobre la tarjeta veremos que el motor gira una vuelta en un sentido, se para dos segundos y vuelve a girar en sentido contrario y así sucesivamente.
Vamos a poner ahora un poco de control sobre el motor paso a paso. Para este segundo ejemplo realizaremos un circuito donde controlaremos el motor paso a paso por medio de un joystick que dependiendo del desplazamiento que presente éste, el motor girará a distintas velocidades y sentidos.
Para el circuito además de los elementos usados en el ejemplo anterior necesitaremos un joystick. Usaremos sólo uno de los ejes por lo que si tenemos un joystick de un solo eje ya nos es válido para el ejemplo; en caso de no tener un joystick podemos usar un potenciómetro ajustando ligeramente el código.
Lo que haremos en el *sketch* será leer el dato proporcionado por el joystick para calcular la velocidad y el sentido de rotación del motor.

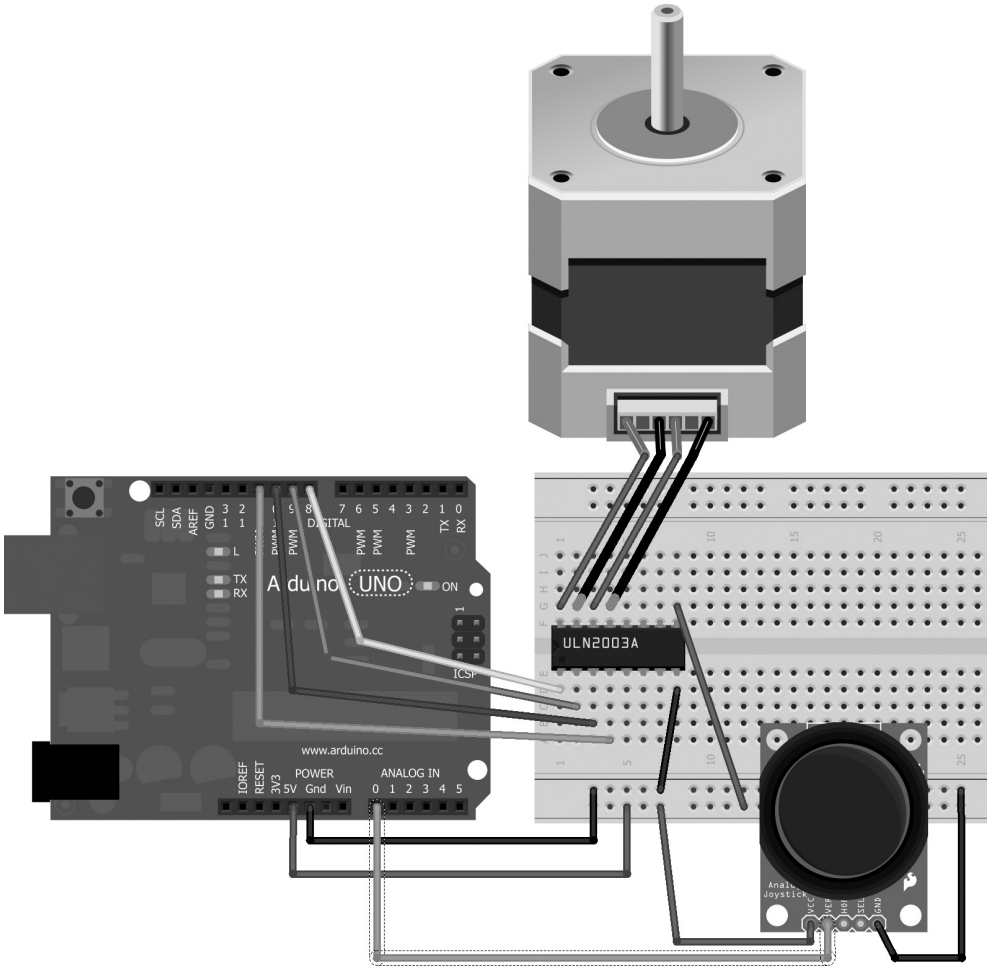


Figura 10.3. Circuito para motor paso a paso controlado por joystick.

Lo primero que necesitaremos para el control mediante el joystick, es habilitar el pin A0 para la lectura de los valores transmitidos por éste.
Necesitaremos también crear una instancia de la clase `Stepper` a fin de controlar los movimientos del motor. Por último necesitaremos que los valores obtenidos de la lectura del joystick, modifiquen los valores de rotación del motor; se ha de tener en cuenta que las velocidades siempre son positivas y que el número de pasos puede ser positivo o negativo, dependiendo de hacia dónde se mueva el joystick y que hará que el motor gire en un sentido o en otro. Como lo que se quiere modificar es la velocidad habrá que transformar el dato presente en A0, que recordaremos es un número de 0 a 1023 en una

velocidad de 10 a 200 rpm; esto lo podemos hacer mapeando los valores obtenidos a una escala de -20 a 20, aplicando el valor absoluto y multiplicando el resultado por 10.

```
int motorSpeed = abs(map(sensorReading, 0, 1023, -20, 20)) *10;
```

La velocidad la ponemos entre 10 y 200 rpm porque velocidades inferiores a 10 son excesivamente lentas y dan un mal efecto. Una vez obtenida la velocidad debemos saber hacia dónde debe girar el motor, para ello miramos si el valor leído es mayor o menor de 511, que es el punto medio y supuestamente es también el valor que debemos obtener si el joystick está en reposo.

```
int rotation = sensorReading>511?1:-1;
```

Si multiplicamos el número de pasos a realizar por la variable `rotation` (que será 1 o -1), tendremos hacia qué lado debe girar ese número de pasos. Ese número de pasos lo definimos mediante una constante, que en este caso serán 5 pasos por cada lectura, es decir, si lee que debe girar a 50 rpm, hará 5 pasos a dicha velocidad y volverá a leer a qué velocidad debe hacer los siguientes 5 pasos. Para mejorar la granularidad se podría reducir el valor de esta constante. El código completo quedaría:

```
#include <Stepper.h>
const byte STEPS = 100;   // Número de pasos por vuelta
const byte STEPS_PER_READ = 5;   // Número de pasos lectura
const int joystickPin = A0;

/*Para conectar el motor necesitamos 4 pines
las entradas In1, In2, In3, In4  se ejecutarán
en secuencia  1-3-2-4  así lo indicamos
en la construcción del objeto*/
Stepper stepper(STEPS, 8, 10, 9, 11);

void setup(){
   pinMode(joystickPin, INPUT);
   stepper.setSpeed(200);
}

void loop(){
   int sensorReading = analogRead(joystickPin);
   // mapeamos de 200 a -200 para las velocidades:
   int motorSpeed = abs(map(sensorReading, 0, 1023, -20, 20)) *10;
   // si existe velocidad hay que mirar el sentido de rotación
   if (motorSpeed > 10) {
      // dependiendo de si la lectura el modificador de rotación
      // será positivo o negativo
      int rotation = sensorReading>511?1:-1;
      // configuramos el motor
      stepper.setSpeed(motorSpeed);
      stepper.step(rotation * 5);
   }
}
```

Se ha añadido la comprobación:

```
if (motorSpeed > 10) {
```

para evitar que si está mal calibrado el joystick, se reciban lecturas próximas al punto medio pero que no sean exactamente el punto medio y que en la transformación mediante `map()` generarían pequeñas velocidades y el motor giraría cuando debería estar parado por estar el joystick en reposo.
Ahora podríamos ya hacer una grúa que girara controlada por el joystick y si usáramos dos motores y en el eje del segundo motor pusiéramos un tornillo sin fin, podríamos convertir el movimiento rotatorio en movimiento lineal y controlar también el avance del carro de la grúa.
El problema que tienen los motores paso a paso es que no se tiene control de la posición real del rotor, se conocen los desplazamientos pero no la posición en sí.
Para subsanar este problema se pueden utilizar discos Gray (véase el apéndice) para posicionamiento mediante sensores de luz o eléctricos o bien usar servomotores.

Servomotores

Los servomotores (también conocidos como simplemente servos) se han hecho un hueco en la electrónica de automatización por la fiabilidad, facilidad de control y bajo coste. Se trata de unos motores que junto con un conjunto de elementos que lo acompañan son capaces de posicionarse de una manera determinada dependiendo de la señal recibida, disponen de un terminal donde enviar una señal que es transformada en una orden acerca de cómo debe moverse y posicionarse el motor.
En el interior de los servomotores podemos encontrar un motor que puede ser de corriente alterna o continua, una caja reductora con engranajes, un potenciómetro y cierta electrónica de control.

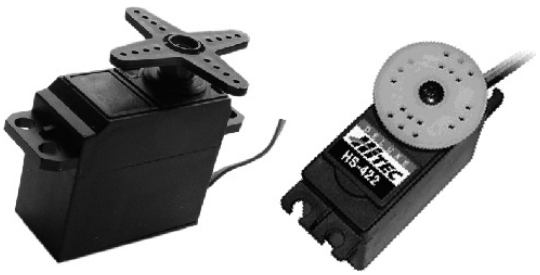


Figura 10.4. Servomotores.

Su funcionamiento básico es el siguiente: El motor se encuentra conectado a la caja de engranajes que se encarga de transmitir el movimiento reducido al brazo del servo, esta caja de engranajes a su vez mueve el potenciómetro que actúa de sensor de posición, cuando se recibe una señal de posición, la transforma en su voltaje equivalente y lo compara con el voltaje obtenido por el potenciómetro y dependiendo de las diferencias entre ellos, el motor girará para equipararlos.

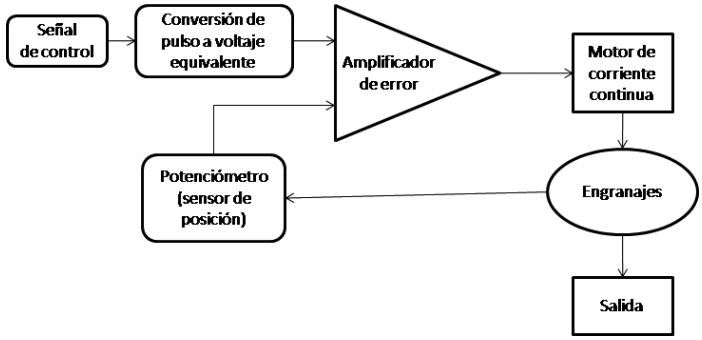


Figura 10.5. Funcionamiento de los servomotores.

El movimiento del brazo del servomotor se indica mediante PWM; cada posición del motor viene dado por un pulso diferente, si el pulso es de1ms, se indica que se debe colocar a 0°, si el pulso es de 2ms se debe colocar a 180° y a partir de estos dos valores se extrapolan el resto, por ejemplo para llevarlo a 90°, el pulso debería ser de 1.5ms.

Normalmente el ciclo completo de pulso son los 20ms, es decir una vez acabado el pulso, tiene de tiempo hasta los 20ms para alcanzar su posición, en este caso tendría 19ms si quisiera alcanzar la posición 0° y 18ms si quiere alcanzar la posición 180°.

Puesto 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° y quiero ir a 0° tengo 19ms para cubrir 1°, pero si quiero ir a °180 tendré 18ms para cubrir 179°. Cada 20ms podremos enviar un pulso con una nueva posición al servo, por lo que podemos cambiar de posición 50 veces por segundo. Véase la figura 10.6.

Vamos a realizar un primer ejemplo en el que haremos que el servo vaya de 0° a 180° en pasos de 1°, vuelva a 0° en pasos de 2° y efectúe un paso brusco de 0° a 180° y viceversa, sin pasos intermedios, así podremos comprobar las diferentes velocidades que puede alcanzar.

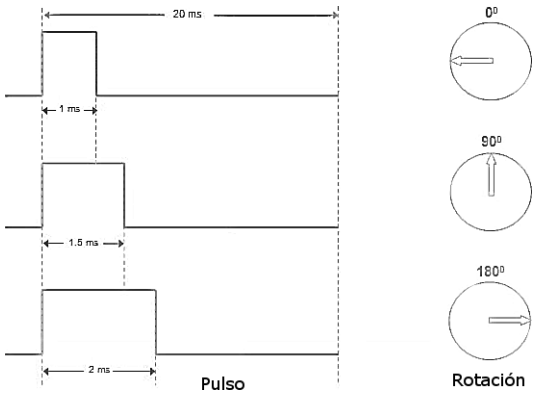


Figura 10.6. Pulsos para indicar posición.

Para el circuito necesitaremos simplemente un servomotor. El servomotor dispone de tres cables, uno que irá a alimentación, otro a masa y el de datos por el que se le informará la posición que debe tomar. El cable de datos debe ir a algún terminal que tenga habilitada la salida PWM, ya que se trabajará utilizando esta técnica para indicarle al servo la posición a la que ir. Recuerde que las salidas habilitadas para PWM vienen marcadas con el símbolo "~" o directamente con la serigrafía PWM.

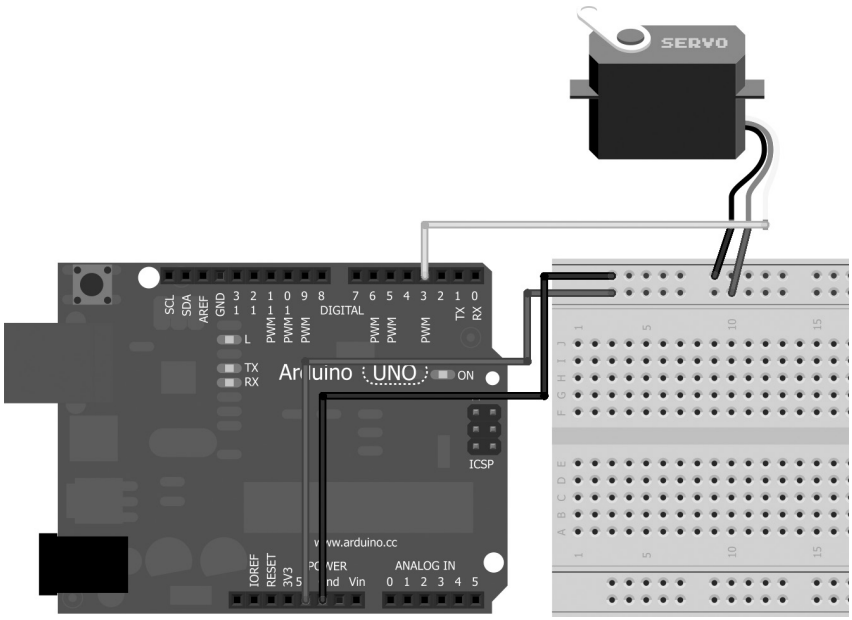


Figura 10.7. Circuito de prueba para servomotor.

Del mismo modo que al trabajar con los motores paso a paso teníamos disponibles unas librerías para facilitar el trabajo, en el caso de los servomotores sucede lo mismo, mediante la inclusión del archivo de cabecera `Servo.h` o la pulsación sobre el menú Sketch>Importar librería...>Servo, se nos hace accesible la clase `Servo` sobre la que podremos trabajar aislándonos de la complejidad de la emisión de pulsos. Para trabajar con el servo, debemos obtener una instancia a la clase y asignarla un servo al que controlar. La asignación se hace mediante el método `attach(numeroPin)`, donde `numeroPin` es el número del terminal Arduino al que se enviarán los pulsos. Para posicionar el servo, simplemente hay que llamar al método `write(grados)` e indicarle los grados a los que queremos que se posicione el servo. Lo más interesante de los servos es que a diferencia de los motores paso a paso podemos conocer la posición en la que se encuentra en cada momento; mediante la llamada `read()` sobre la instancia de la clase se obtiene la posición en la que se encuentre el servo en ese momento; esta posición viene dada en grados de 0 a 180. Si se quiere dejar de utilizar el servo se puede llamar a la función `detach()` y saber si está unido o no por medio de la llamada `attached()` que devolverá un booleano indicando el estado de la unión. Todas estas llamadas se deben realizar sobre una instancia de la clase Servo.

Para realizar el movimiento indicado anteriormente (que vaya de 0° a 180° en pasos de 1°, vuelva a 0° en pasos de 2° y efectúe un paso brusco de 0° a 180° y viceversa) el *sketch* sería:

```
#include <Servo.h>

Servo servo;  // crea la instancia de la clase

int servoPos = 0;    // posición del servo

const byte  servoPin = 3;

void setup(){
   servo.attach(servoPin);  // Se une el servo con el pin 3
}
void loop(){
   // llevamos de 0 a 180° en grado a grado
   for(servoPos = 0; servoPos < 180; servoPos ++){
      servo.write(servoPos);
      delay(20);            // tiempo de espera
   }
   // volvemos a 0 un poco más rápico
   for(servoPos = 180; servoPos>=1; servoPos-=2){
      servo.write(servoPos);
      delay(20);            // tiempo espera
   }
   // barrido rápido de 0 a 180 y vuelta
   servo.write(180);
```

```
    delay (500);
    servo.write(0);
    delay (500);
}
```

Muy importante son los `delay(20)` que hay en los bucles, ya que si no estuvieran, el bucle 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, por lo tanto estaríamos perdiendo información.

Nota:

Un ciclo completo del servomotor se completa en 20ms.

Vamos a hacer ahora algo semejante a lo realizado con el motor paso a paso, vamos a controlar su posición mediante un potenciómetro. Lo que haremos será leer la entrada de dicho potenciómetro y ajustarla a los valores esperados por el servo. Como entrada de los valores del potenciómetro se usará la entrada analógica A0.
En el *sketch* de control de este circuito añadiremos además información sobre la posición en la que está el servo en cada momento; esta información la mostraremos mediante el monitor serie.

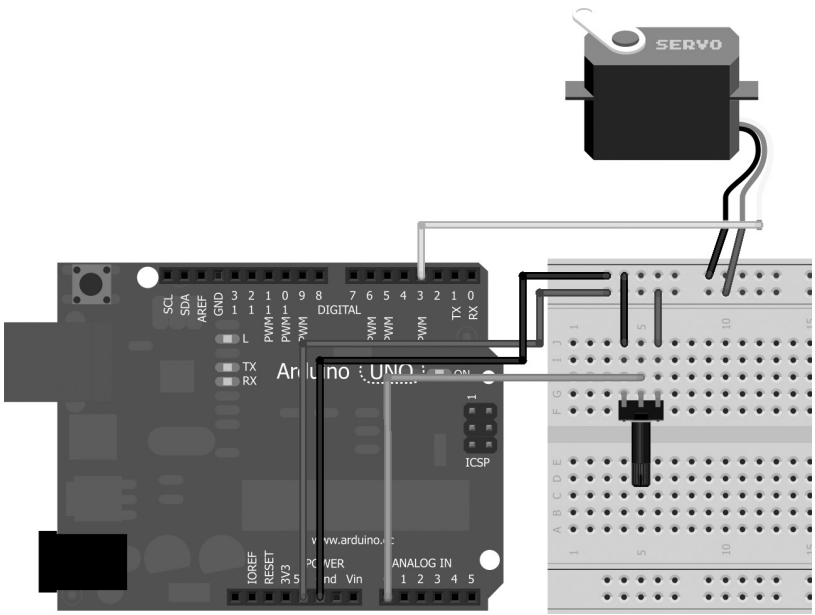


Figura 10.8. Circuito de prueba para servomotor con potenciómetro.

Necesitaremos entonces definir el pin por el que se leerá el potenciómetro y el pin del servomotor. Una vez leído el dato del potenciómetro hay que mapearlo a valores comprendidos entre 0° y 180° y mostrarlo en pantalla; a modo didáctico, tras escribir la posición en el servo, se recupera para volverla a mostrar en pantalla.

```
#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);
   }
}
```

Dado que leemos directamente el valor del potenciómetro es muy probable que existan fluctuaciones de los valores en la entrada dados por, por ejemplo, la calidad del potenciómetro u otros sistemas electrónicos que tengamos cerca y afecten a la señal, lo que hará que en ocasiones de por ejemplo de entrada 1000 y sin tocar el potenciómetro otras veces de 1010; esto implica que varíen los grados del servo. Se podría estabilizar el valor de entrada mediante un circuito auxiliar, pero en nuestro caso lo haremos algo más sencillo y es simplemente evitar mover el servo si el valor recibido es menor de 5 grados. Esto se realiza mediante:

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

donde se comprueba si la última lectura del servo difiere al menos en 5 grados respecto de la que debe tener y si es así, modifica la posición del servo; si no lo es, desecha la lectura.

Si probamos el circuito podremos ver cómo según se mueve el potenciómetro se va moviendo también el brazo del servomotor dependiendo de la posición de aquel.

Relés

Los relés unos elementos que se utilizan en los circuitos a modo de interruptores pero a diferencia de los que hemos visto hasta ahora, estos se activan mediante corrientes eléctricas. Existen de múltiples tipos, utilizando diferentes tecnologías para conseguirla apertura o cierre de circuito; unos de los más comunes son los denominados de armadura; este tipo de relés se basa en la atracción por parte de un campo electromagnético de una chapita que se encarga de cerrar el circuito controlado cuando en el circuito controlador existe excitación eléctrica suficiente.
En un relé tenemos principalmente dos circuitos, el de control del propio relé y el controlado por el relé. Los relés separan completamente el circuito de control y el circuito controlado por lo que permiten tener dos circuitos de naturaleza completamente distinta, por ejemplo podemos controlar con un relé de corriente continua y 5V un circuito de corriente alterna de potencia (siempre que el relé lo soporte).

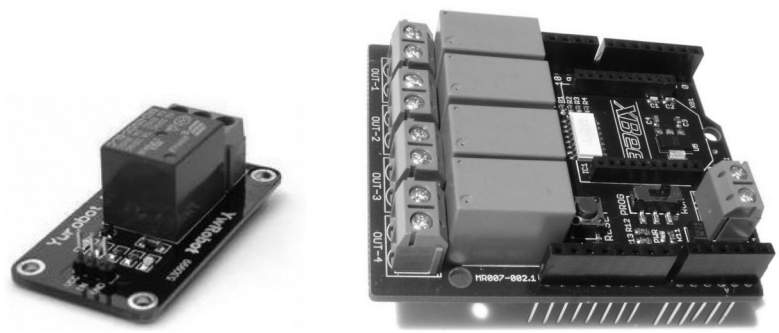


Figura 10.9. Relés.

Podemos también clasificarlos en relés en los que siempre hay un circuito cerrado y los que conectan y desconectan el circuito. Los que conectan y desconectan el circuito suelen tener dos entradas para el circuito a controlar y cuando se le da la señal de activación al relé, los terminales quedan unidos cerrando el circuito y en caso contrario el circuito estará abierto; por contra el otro tipo de relés tiene tres bornes, uno común y los otros dos se encargarán

de cerrar dos circuitos, uno de los circuitos se cerrará cuando el relé no reciba señal y se abrirá cuando la reciba y el otro se actuará de forma contraria, abriéndose cuando no haya señal y cerrándose cuando la haya. El borne que tiene el circuito cerrado en ausencia de excitación 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 serigrafías en los relés; claro está que el borne que no posea serigrafía será por descarte el común. En el esquema de la figura 10.10 podemos verlo más claramente.

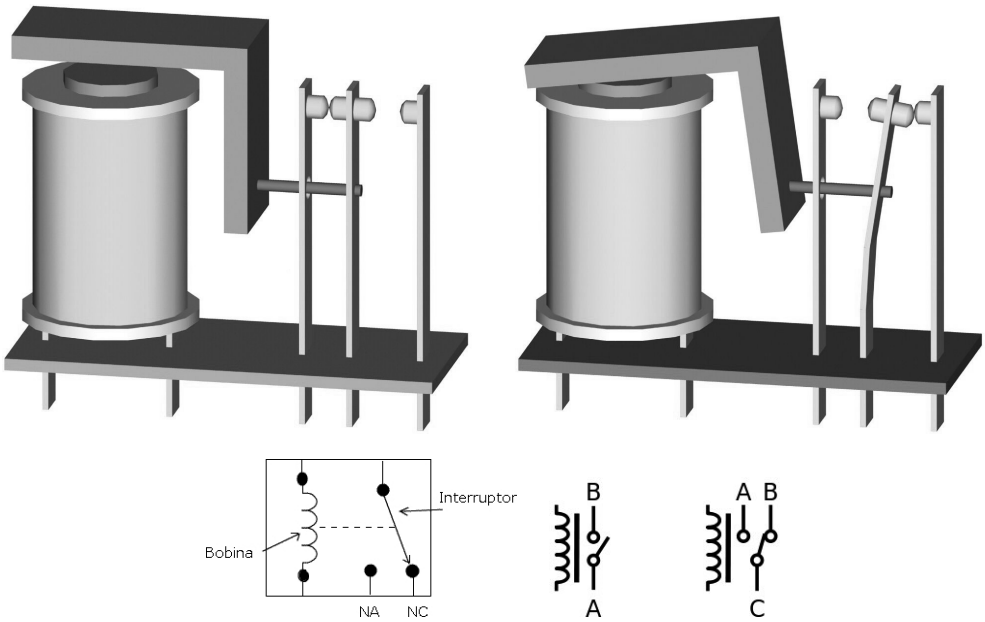


Figura 10.10. Esquemas de funcionamiento de un relé.

En las tiendas podemos encontrar relés sueltos (de una sola bobina), grupos de relés, *shields*... y los podemos encontrar que el circuito de control sea a 5V, a 12V, etc.. Con esto quiero decir que antes de hacer un circuito se debe mirar qué es lo que se va a controlar y con qué se va a controlar. En caso de tener un relé con control a 12V, no lo podríamos conectar directamente con Arduino (ya que sólo disponemos de terminales hasta 5V) y necesitaríamos una fuente externa para su control, lo mismo sucede con la intensidad, dependiendo de los amperios que necesite el relé para su activación necesitaremos fuente externa de alimentación. En caso de usar una fuente externa, se puede realizar el montaje añadiendo un optoacoplador que nos servirá para aislar

los circuitos y activar el relé. Algunas *shields* de Arduino traen varios relés preparados para trabajar con fuentes de alimentación externas sin necesidad de añadir nada.

Quizá los más comunes sean los relés con tres bornes en el circuito secundario, así que veremos cómo conectarlos. En el circuito primario o de control tenemos dos o tres bornes, uno de ellos irá a tensión, el otro a masa y el otro a señal (dependiendo de los relés, el borne de señal y el de tensión es el mismo). Mientras no haya tensión en el borne de señal, la plaquita de metal interna permanecerá apoyada en el borne NC, por lo que conectaremos al común y al NC lo que queremos que funcione mientras no haya señal. En el momento que se da voltaje en el borne de señal del relé, se activa el electroimán atrayendo la chapa interna, abriendo el circuito que estaba cerrado y cerrando el circuito que estaba abierto, correspondientes a los bornes común y NA (NO); entonces conectaremos al común y a NO lo que queremos que se conecte cuando esté el relé conectado. Si no interesa conectar algo cuando esté el relé excitado o cuando no lo esté, se puede dejar alguno de los bornes NO o NC al aire, simplemente no se conectará nada cuando se cierre el circuito.

Lo más divertido de los relés es que con muy poco esfuerzo podemos hacer que interactúen con elementos cotidianos, por ejemplo si tenemos un niño pequeño podemos hacer un circuito con un temporizador unido a un relé y que pasado un tiempo determinado se desactive y a pague la lámpara de noche o añadirle un micrófono y con su ayuda detectar si llora, al pasar la señal del micrófono cierto umbral, y que active automáticamente de nuevo el relé encendiendo la lámpara. Veamos cómo utilizarlos.

Para el siguiente ejemplo necesitaremos un relé, una fotorresistencia, un led y una resistencia de 220Ω y otra de 10kΩ para el divisor de tensión de la fotorresistencia. Véase la figura 10.11.

Vamos a realizar un circuito que haga que cuando haya poca luz se encienda una lámpara. La falta de luz la detectaremos mediante lecturas al divisor de tensión de la fotorresistencia, tal y como lo hicimos en el capítulo 7, si estas lecturas superan un umbral, entonces mandaremos la señal al relé para que cierre el circuito NA. Por la parte del circuito secundario del relé, el controlado, hemos realizado un montaje con un led como los hemos hecho ya varias veces, donde una de sus ramas se encuentra interrumpida por el relé, de modo que cuando el NA y el común del relé se cierren, se cerrará todo el circuito del led y éste lucirá.

El *sketch* es muy sencillo pero efectivo, simplemente se configuran los pines para entrada (fotorresistencia) y salida (relé) y en el bucle principal se debe ir comparando la lectura de la entrada de la fotorresistencia con el umbral que queramos usar, que en el ejemplo es de 800.

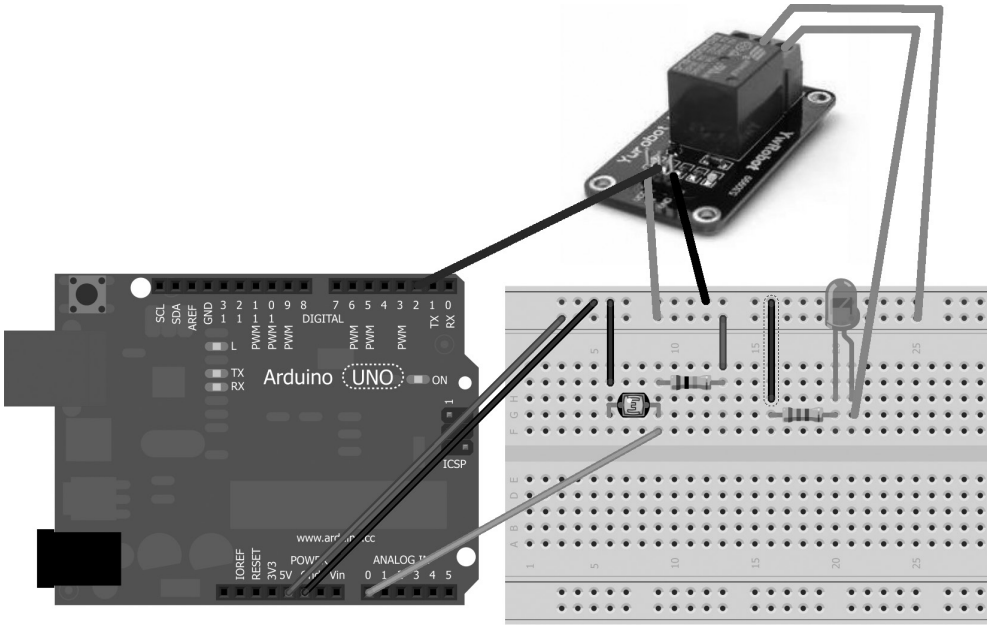


Figura 10.11. Circuito con relé.

Si la lectura es mayor a 800 quiere decir que falta luz, por lo que hay que encender la bombilla y se envía un `HIGH` al pin de señal del relé y en caso de tener una lectura por debajo de 800 quiere decir que hay luz así que enviamos un `LOW` al relé.

```
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);
}
```

En este caso se ha utilizado un circuito secundario con led, pero se podría haber usado una lámpara cotidiana, de las de 220 voltios (siempre que el relé lo soporte), de modo que al haber poca luz se encendiera dicha lámpara en lugar de el led; la manera de actuar es semejante, "interrumpir" una de las ramas de alimentación a la bombilla poniendo en medio los bornes común y NA el relé. Si se opta por realizar éste montaje (es más espectacular y siempre se puede presumir de tener una lámpara que se activa sola) se debe tener en cuenta que cuando se encienda la luz, la fotorresistencia volverá a tener valores bajos al recibir la luz lo que hará que se apague de nuevo la luz, por lo que deberemos controlar que no se esté encendiendo y apagando lámpara continuamente, por ejemplo seleccionando un emplazamiento correcto de la fotorresistencia donde se vea afectada por la luz ambiente pero no por la de la lámpara. Otro montaje que se podría hacer sin apenas esfuerzo y con tan sólo un micrófono sería una lámpara controlada por sonido, que por ejemplo se active o desactive al detectar tres palmadas o ser controlada por infrarrojos y utilizar el mando a distancia de la televisión... ¿Se anima?

11 Bluetooth

En este capítulo aprenderá a:

- Descubrir los fundamentos de Bluetooth.
- Conocer una nueva manera de comunicarse.
- Controlar la tarjeta Arduino a distancia.
- Realizar emisión de datos hacia la tarjeta Arduino.
- Realizar emisión de datos desde la tarjeta Arduino.

En el día a día cuando utilizamos componentes electrónicos, normalmente no los usamos solos, sino que utilizamos periféricos para aumentar su usabilidad o prestaciones; si pensamos por ejemplo en el ordenador, es totalmente inútil sin un teclado, una pantalla y un ratón. Las comunicaciones con los periféricos pueden darse de muchas formas, mediante cables serie, WiFi... o Bluetooth. Las conexiones Bluetooth están proliferando cada día más debido a su simplicidad y su eficiencia y las podemos encontrar en marcos digitales, auriculares, ratones inalámbricos, teléfonos móviles, manos libres, GPS... en infinidad de dispositivos.
Cuando se quiere conectar dos dispositivos surgen una serie de problemas a solucionar, por ejemplo:

- ¿Cuántos datos se enviarán en cada paquete?, ¿será serie o paralelo?, ¿con qué frecuencia?
- Si utilizan cables... ¿cuántos usarán?, ¿a qué voltaje?, ¿con qué numeración de pines?
- ¿Qué protocolo se usará? ¿Qué comandos?
- ...

Existen multitud de preguntas a las que contestar para que un sistema de comunicación entre dos dispositivos llegue a buen puerto.
En el caso de Bluetooth muchas de ellas ya vienen respuestas: trabaja sin cables, con frecuencias de radio en la banda de los 2.4GHz (entre 2.402 GHz y 2.480 GHz en concreto), con un protocolo de comunicación preestablecido por las partes que deben consensuarlo durante la transmisión de los bits.
La banda de los 2.4GHz es también la de las comunicaciones WiFi, pero a diferencia de éstas, Bluetooth apenas usa 1mW de potencia para la comunicación, es decir es muy cuidadoso con las baterías (aunque no lo parezca). El hecho de trabajar con tan poca potencia hace que su radio de actuación sea mucho más pequeño que el de la WiFi, siendo como mucho de 100 metros, lo cual también es una ventaja ya que crea lo que se llama una PAN (*Personal Area Network*, Red de área personal) o piconet, una especie de red privada con altos niveles de seguridad entre los dispositivos de modo que la información no se divulgue más allá. Dependiendo de la potencia consumida por Bluetooth se clasifican en tres clases:

- Clase 1: Consume una potencia de 100mW y tiene un alcance cercano a los 100m.
- Clase 2: Con un consumo de 2.5mW, tienen un alcance de unos 20m.
- Clase 3: Tan solo consumiendo 1mW da un alcance de unos 2m.

Con lo que podemos ver que son transmisiones de muy baja potencia, en comparación con las de telefonía móvil que son de alrededor de 3W.
Por lo que llevamos visto sobre Bluetooth se hace ideal para transmisiones de corto alcance entre dispositivos, pero ya existía la comunicación mediante infrarrojos, ¿qué aporta esta tecnología de nuevo?
La comunicación por infrarrojos está presente en los dispositivos electrónicos desde hace mucho tiempo y tiene muchas ventajas frente a Bluetooth en algunos aspectos, por ejemplo en el precio, ya que el hardware de comunicación por infrarrojos es mucho menor; pero en otros aspectos no. Los mandos de la televisión siguen funcionando mediante rayos infrarrojos, pero tenemos la peculiaridad que normalmente vemos la televisión sentados frente a ella y para, por ejemplo, cambiar de canal apuntamos con el mando a distancia y cambiamos... pero ¿qué pasa si no apuntamos a la televisión? Pues que no suele funcionar. La comunicación mediante infrarrojos es lineal, la luz se emite desde un led que normalmente suele estar como incrustado en un cono para dirigir el haz hacia adelante y se deben colocar en línea y sin obstáculos el receptor y el emisor mientras que con Bluetooth podemos traspasar paredes sin ninguna dificultad al ser ondas de radio en lugar de ondas lumínicas. Por otro lado las tasas de transmisión son mucho mayores en Bluetooth, llegando a 25Mbps en la versión 3.0 y 4.0. Por último, otra diferencia es que la transmisión usando infrarrojos suele ser entre dos dispositivos, mientras que con Bluetooth podemos llegar a hacer transmisiones con múltiples dispositivos.

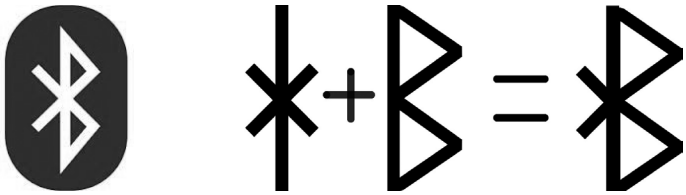


Figura 11.1. Logo de Bluetooth y runas.

El hecho de que Bluetooth transmita en la frecuencias de 2.4GHz, provoca que pueda interferir con múltiples dispositivos que también actúan a estas frecuencias, como pueden ser mandos de coche o bien de garaje, monitores de bebés, altavoces... pero dado que su potencia de transmisión es muy baja y su alcance muy corto es realmente extraño que se puedan llegar a dar estas interferencias.
Según el estándar, a un dispositivo Bluetooth se pueden conectar hasta 8 dispositivos a la vez, pero si decimos que tenemos un radio de acción muy corto por la pequeña potencia con la que se emite, esos 8 dispositivos deben

estar muy cerca del dispositivo central Bluetooth y se generarían interferencias entre ellos; para evitar este caso se utiliza una técnica llamada FHSS (*Frequency-hopping spread spectrum*, salto de frecuencia en ancho de espectro) que consiste en que el emisor y el receptor cambian la frecuencia a la que transmiten cada cierto tiempo de manera pseudo aleatoria y conocida tanto por el emisor como por el receptor; en Bluetooth, los dispositivos poseen 79 frecuencias distintas sobre las que transmitir y el cambio entre ellas se realiza hasta en 1600 ocasiones por segundo, lo que hace que sea muy difícil que dos dispositivos estén en la misma frecuencia en el mismo tiempo.

Cuando dos dispositivos Bluetooth se encuentran en un rango donde son visibles mutuamente, comienza una conversación entre ellos para determinar si ya son conocidos y si tienen datos a intercambiar y en caso de ser así, la comunicación se produce sin necesidad de intervención por parte del usuario; así, cuando entramos en el coche con el móvil, el Bluetooth del manos libres se conecta sin tener que hacer nada especial, creando una red interna entre ellos, una piconet. Si dentro del mismo coche hubiera alguien con un reproductor de DVD y unos auriculares conectados a ellos por Bluetooth, no habría problemas de interferencias, ya que cada uno tendría su propia piconet y trabajarían a frecuencias distintas y en caso de coincidir alguna señal en una de las frecuencias, al no pertenecer a la red, ésta se desecharía con lo que no produciría tampoco interferencia.

La seguridad es este tipo de comunicaciones es un punto a tener en cuenta, ya que al tratarse de ondas de radio que pueden ser captadas no linealmente; podrían ser escuchadas por dispositivos no autorizados. Supongamos que estamos en un bar tomando un café y estamos transmitiendo datos de nuestro móvil al ordenador mediante Bluetooth; en la mesa de al lado podría haber una persona con otro dispositivo Bluetooth escuchando y al estar tan cerca nuestro estaría dentro del rango de nuestra piconet y por tanto con acceso a ella a nivel físico, además con la particularidad de conexión automática que tiene Bluetooth, sería realmente fácil escuchar los datos enviados sin permiso. Por todo esto Bluetooth ofrece múltiples modos de seguridad y cada fabricante selecciona cual es el que quiere utilizar.

Algunos ejemplos son:

- Estableciendo que el dispositivo no sea descubrible de modo que cuando hacemos buscar dispositivos Bluetooth no aparezca (aunque realmente sí que esté).
- Determinando dispositivos seguros por identificador de modo que cuando se conectan se revisa el identificador del dispositivo que se intenta conectar y solamente si se encuentra en una lista del receptor podrá conectarse.

- Mediante pin; donde al intentar conectarse, el dispositivo receptor mira a ver si ya se ha conectado anteriormente y se le ha dado paso, en caso contrario establece un pin de conexión que debe ser igual por las dos partes; desde ese momento el dispositivo puede pasar a estar emparejado permanentemente o que durante la siguiente comunicación le vuelva a pedir el pin, esto depende de la implementación del fabricante.

Aún y con todas las medidas de seguridad, Bluetooth puede ser vulnerable, por lo que si se va a transmitir información confidencial, es mejor hacerlo de modo cifrado.

Entonces hasta ahora conocemos que la tecnología Bluetooth es un modo de transmisión inalámbrico que opera en la banda de los 2.4GHz, con muy poca potencia de transmisión y corto alcance y que puede añadírsele seguridad, pero ¿por qué en ocasiones algunos dispositivos aceptan por ejemplo los auriculares Bluetooth y otros no? ¿O porqué pese a tener Bluetooth desde el principio, los primeros iPhone no permitían transmitir ficheros entre ellos? Esto se debe a los perfiles Bluetooth. Este tipo de perfiles son unas especificaciones sobre cómo se debe interpretar los datos enviados, eso sí, siempre basándose en el núcleo de la especificación Bluetooth.

Para que dos dispositivos puedan trabajar emparejados deben ser compatibles con el set de perfiles Bluetooth necesarios para completar la tarea, dicho de otro modo, si queremos recibir audio, deberá ser compatible con el perfil A2DP (*Advanced Audio Distribution Profile*, Perfil Avanzado de Distribución de Audio) mientras que no es necesario que cumpla el perfil BPP (*Basic Printing Profile, Perfil Básico de Impresión)*; en cambio una impresora sí lo deberá cumplir (y el dispositivo que se conecte a ella). Las capacidades de cada dispositivo dependen de los perfiles que soporte (aparte de la versión de Bluetooth implementada), pero eso no quiere decir que tengamos que tener obligatoriamente soportado un perfil, simplemente que si queremos hacer las cosas más estándares se deberían soportar los perfiles ya establecidos, lo cual no quita (como veremos más adelante) que podamos hacer transmisiones sin necesidad de cumplir ningún perfil; lo único que tenemos que asegurar es que el emisor y el receptor entiendan los datos transmitidos.

En cuanto a Arduino se refiere, en el mercado encontraremos desde módulos de Bluetooth simples y muy baratos hasta *shields* complejas con Bluetooth y otros sensores incorporados. Véase la figura 11.2.

Después de tanto dato técnico y antes de entrar a realizar algunos ejemplos vamos a ver un poco de historia de Bluetooth, ya que es muy particular y se entenderá el porqué de la figura 11.1. La tecnología detrás de Bluetooth data de los años 40; como otra de tantas tecnologías, nació en el seno militar,

pero fue en 1994 cuando la compañía sueca Ericsson implementó esta tecnología para la eliminación del cable en las comunicaciones RS232. Será en 1998 cuando varias compañías de gran calado (como IBM o NOKIA) muestren interés por esta tecnología y se crea el SIG (*Bluetooth Special Interest Group*) que es una organización que se encarga de desarrollar los estándares de Bluetooth y velar por el cumplimiento de ellos a la vez que gestionar las licencias a los fabricantes de dispositivos.

El nombre de Bluetooth proviene del rey Danés Harald Blåtand que vivió a finales de los 900s; este rey consiguió unir varias tribus escandinavas entre las que se encontraban los daneses y gran parte de los noruegos y convertirlos al cristianismo. Harald I murió en 986 durante una guerra con su propio hijo. La versión inglesa del nombre Harald Blåtand es Harald Bluetooth y dado que intentó unificar pueblos, se propuso su nombre para esta nueva forma de comunicación que une dispositivos heterogéneos. El logo viene de las runas usadas para formar sus iniciales HB, la runa Hagal ᚼ y la runa Bjarken ᛒ.

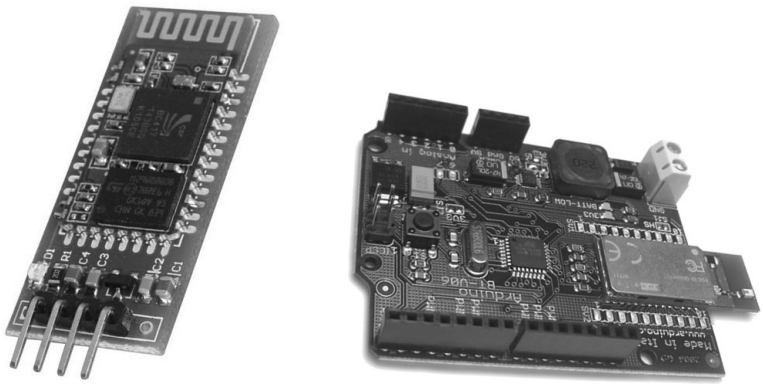


Figura 11.2. Módulo y *shield* Bluetooth.

Control mediante Bluetooth

Existen múltiples dispositivos controlados mediante Bluetooth, por ejemplo los ordenadores mediante ratón inalámbrico o los helicópteros de juguete que se han puesto tan de moda y que se pueden controlar desde los teléfonos móviles. El funcionamiento es sencillo; el dispositivo móvil se empareja con el dispositivo a controlar y a partir de ese momento el móvil simplemente tiene que enviar hacia el dispositivo controlado algún tipo de comando que pueda ser interpretado.

Para la interpretación de comandos no hace falta cumplir ningún perfil específico siempre que tanto el emisor como el receptor entiendan el comando que se está enviando. Haciendo un símil con el lenguaje humano, si hacemos un libro que esté en chino, nos aseguramos que más de 1300 millones de personas lo entenderán, pero si nos inventamos un idioma nosotros y nos aseguramos que la persona que lo va a leer sepa el idioma, el problema está resuelto aunque no sería un *best seller* porque nadie lo entendería; en este caso escribir en chino sería usar algún perfil de Bluetooth, usar un estándar para que muchos puedan usarlo e inventarnos el idioma sería enviar unos códigos inventados por nosotros que hacemos que el receptor sepa interpretar.
Para este primer ejemplo vamos a hacer que desde un dispositivo móvil y mediante Bluetooth podamos encender y apagar diferentes leds. El dispositivo móvil empleado para el ejemplo será un Android, pero puede ser adaptado a cualquier otro sistema operativo incluso a PC, simplemente debe enviar los mismos comandos.
Para el circuito de este ejemplo necesitaremos tres resistencias de 220Ω, tres leds, un módulo de transmisión Bluetooth y para el funcionamiento total también será necesario un dispositivo móvil Android (tableta o teléfono) que se encargará de enviar los comandos.
La mayor parte de las conexiones del circuito son ya de sobras conocidas; se tratan de tres leds con sus correspondientes resistencias limitadoras y que estarán controlados por los pines 4, 5 y 6 de la tarjeta Arduino, de modo que cuando el receptor obtenga el comando para encender o bien apagar cada uno de ellos, simplemente se tenga que derivar el valor correspondiente hacia el pin concreto. Por otro lado la conexión del módulo de Bluetooth depende mucho de la marca, pero normalmente vale con alimentar el módulo y conectar los cables de transmisión y recepción, marcados como TX y RX respectivamente a los pines 2 y 3 de la tarjeta Arduino que serán los que usemos en el ejemplo. Por si el lector quiere ahorrar un cable, realmente para este ejemplo valdría simplemente con conectar el cable de recepción, ya que no se va a transmitir nada desde Arduino hacia el dispositivo.
El *sketch* a utilizar deberá leer los datos recibidos por el módulo que está conectado a los terminales 2 y 3, pero hasta ahora hemos trabajado siempre con los pines 0 y 1 para la transmisión y recepción de datos, así que necesitaremos echar mano de una librería que nos ayude a poder leer estos datos.
Anteriormente habíamos comentado que la tarjeta Uno simplemente tiene dos pines para puerto de transmisión serie (el 0 y el 1) pero que en el caso de ser necesario, por ejemplo varias fuentes de transmisión, se podrían utilizar otros para tal fin; este es el caso. Si se usan tarjetas con más pines preparados para la transmisión serie, no sería necesaria la utilización de la librería y se

podrían usar los otros puertos serie estándar. La librería que utilizaremos es la *SoftwareSerial*, para ello como en otras ocasiones la añadiremos mediante la sentencia `include` correspondiente. Al importar esta librería podremos crear una instancia de la clase `SoftwareSerial` que funcionará del mismo que el objeto `Serial`, con la diferencia que en este caso estamos utilizando unos pines diferentes. Para generar la instancia se utiliza la nomenclatura:

```
SoftwareSerial nombreInstancia(pinTransmisión, pinRecepción);
```

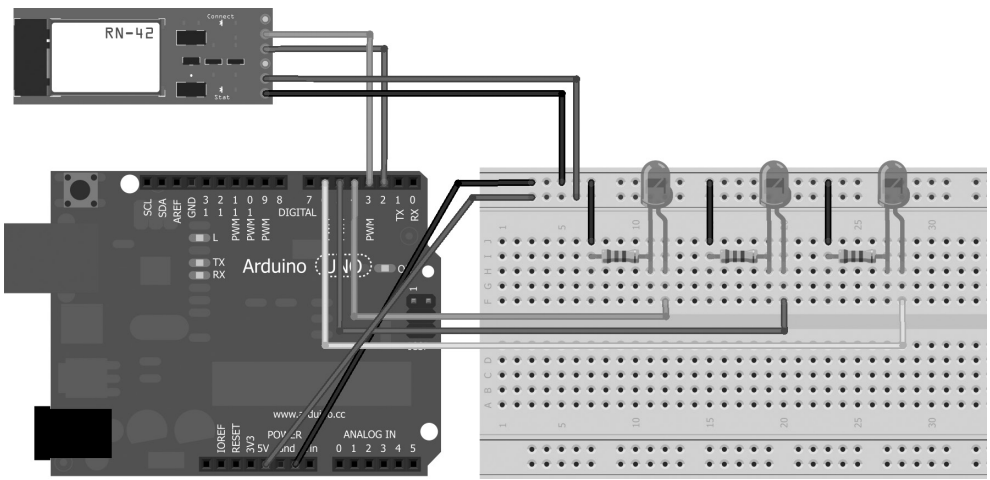


Figura 11.3. Circuito de leds controlado por Bluetooth.

Durante la función de `setup()`, se deben configurar los pines de salida de los leds y la conexión al Bluetooth. Para manejar mejor los pines correspondientes a los leds, los guardaremos en un array, de modo que los podamos recorrer fácilmente:

```
void setup(){
   // Realizamos la conexión Bluetooth con el dispositivo móvil
   bluetooth.begin(9600);
   for (int i =0; i<3; i++){
      pinMode(ledPin[i],OUTPUT);
      digitalWrite(ledPin[i],LOW);
   }
}
```

Durante el bucle principal leeremos la entrada del puerto serie del Bluetooth del mismo modo que hemos trabajado anteriormente con el `Serial` y una vez obtenido el dato se debe interpretar. Vamos a tomar como protocolo, que en la comunicación con el control remoto, éste enviará dos bytes, uno conteniendo el led a controlar y otro con el estado a poner el led. Como tenemos

tres leds, los posibles valores del primer byte son 0 1 y 2 y para el segundo byte vamos a tomar como posibles valores L (de `LOW`) para cuando queramos apagarlo y H (de `HIGH`) para cuando queramos encenderlo y los números se enviarán como caracteres, con lo que ello implica. Antes de proseguir vamos a recordar que cuando los números se envían como caracteres o cuando se envían letras, los bytes enviarán el número ASCII de la representación del carácter, con lo que no recibiremos en el byte un 1 sino su número ASCII que es 49 y así con todos. La tabla de transformación es:

Carácter	Código ASCII
0	48
1	49
2	50
H	72
L	76

Así que lo que haremos es obtener el primer byte que corresponde al led a controlar y guardar su valor, acto seguido obtener el segundo byte que será el estado al que se tiene que poner el led y aplicar el estado al led seleccionado.

```
if(bluetooth.available()){
   byte receviedChar = bluetooth.read();
   // obtenido el byte, comprobar qué se debe hacer con él
}
```

El *sketch* completo sería:

```
#include <SoftwareSerial.h>
const byte btTx = 2;
const byte btRx = 3;
byte ledIndex = -1; // índice de led a controlar
const byte ledPin[] = {4,5,6};
SoftwareSerial bluetooth(btTx, btRx); // instancia serie para Bluetooth

void setup(){
   Serial.begin(9600);
   // Realizamos la conexión Bluetooth con el dispositivo móvil
   bluetooth.begin(9600);
   for (int i =0; i<3; i++){
      pinMode(ledPin[i],OUTPUT);
      digitalWrite(ledPin[i],LOW);
   }
}

void loop(){
```

```
   // obtenemos lectura de Bluetooth
   if(bluetooth.available()){
      byte receviedChar = bluetooth.read();
      if (receviedChar == 48){ //0
         ledIndex = 0;
      }
      else if (receviedChar == 49 ){//1
         ledIndex = 1;
      }
      else if (receviedChar == 50){//2
         ledIndex = 2;
      }
      else if (receviedChar == 72){ //H
         writeLed(HIGH);
      }
      else if (receviedChar == 76){ //L
         writeLed(LOW);
      }
   }
}

void writeLed(boolean level){
   digitalWrite(ledPin[ledIndex], level);
}
```

Ya que se han guardado los pines correspondientes a cada uno de los leds en un array, nos es muy sencillo guardar el índice enviado y cuando se obtenga el estado al que se debe estar, ponerlo directamente mediante la sentencia.

```
digitalWrite(ledPin[ledIndex], level);
```

En este momento la parte Arduino ya estaría totalmente operativa, ahora deberíamos ocuparnos del dispositivo que se utilizará como cliente de este circuito. Lo primero que debemos hacer en caso de que no se haya hecho previamente, es emparejar el dispositivo Bluetooth con el dispositivo Android. Para ello, con el circuito Arduino correctamente configurado y con tensión (que funcione el Bluetooth), debemos dirigirnos a la parte de configuración de Android para poder emparejarlo. Hay que ver que la opción Bluetooth se encuentra activada, y tras ello pulsamos sobre la palabra Bluetooth para entrar a la búsqueda de dispositivos. Al pulsar sobre el botón **Buscar** y tras unos instantes, se mostrará una lista de dispositivos disponibles, en esta lista debería aparecer el módulo sobre el que estamos trabajando (nos guardaremos este nombre porque más adelante lo usaremos); al pulsar sobre él se nos pide un pin para el emparejamiento, que normalmente suele ser 1234. Una vez introducido el pin, el dispositivo y el módulo Bluetooth de Arduino estarán emparejados y tendrán una conexión de confianza, de modo que cuando exista una petición de conexión por parte del dispositivo Android de conectarse, el módulo le dará paso.

Figura 11.4. Pasos para el emparejamiento del Bluetooth.

Para la parte de programación vamos a necesitar el entorno de desarrollo de Android, disponible en la Web de Android (`http://developer.android.com/sdk/index.html`).
Una vez descargado y descomprimido el entorno, lo abriremos y descargaremos los elementos necesarios según la guía de instalación que lo acompaña.
Luego entraremos en la parte de codificación de la aplicación Android, se darán los pasos y el código fuente necesario para que la aplicación funcione correctamente pero no se entrará en detalle; puede obtener más información sobre cómo realizar aplicaciones Android en el libro "*Desarrollo de Aplicaciones para Android*" de esta misma colección y autor.
Cuando tengamos todo preparado crearemos un nuevo proyecto Android. Al crear el proyecto se nos preguntará por una serie de parámetros que nos servirán para configurar la aplicación, los cumplimentaremos de este modo:

- Application Name: BluetoothControl.
- Project Name: BluetoothControl.
- Package Name: com.acme.bluetoothcontrol.
- Minimum Required SDK: API 14.

Una vez completados estos campos podemos continuar los siguientes pasos sin modificar las opciones seleccionadas por defecto, si bien se debe comprobar que esté seleccionada la opción BlankActivity en el paso correspondiente a la creación de actividades, ya que en alguna versión no aparece seleccionada por defecto y nos ahorrará mucho trabajo de borrar código.

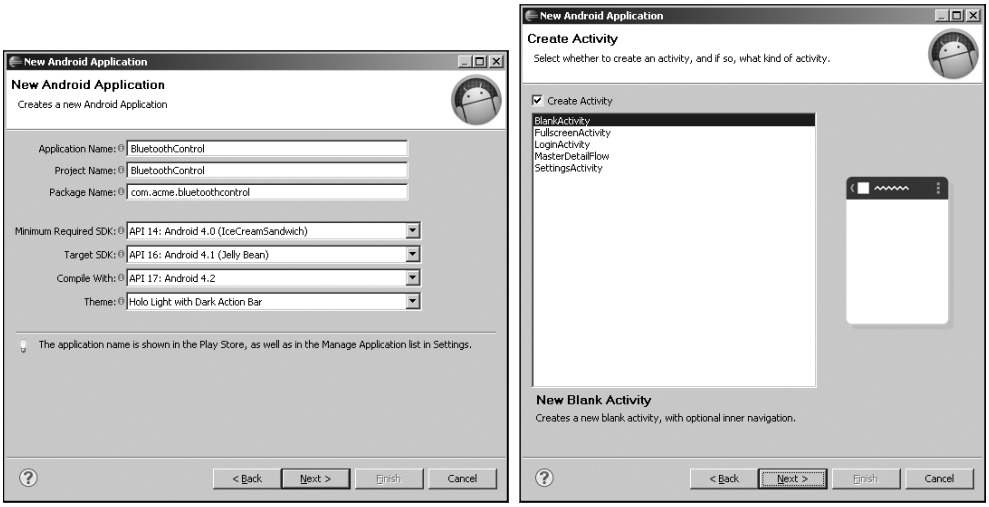


Figura 11.5. Configuración del proyecto Android.

La aplicación que realizaremos en Android mostrará un botón en la parte superior para conectarnos al Bluetooth y tres botones que mostrarán el estado de los tres leds del circuito Arduino y al pulsar sobre ellos (siempre que exista conexión con el Bluetooth), se enviarán los datos correspondientes. Para modificar la interfaz gráfica nos dirigimos a en la estructura del proyecto al fichero situado en la carpeta `res/layout` y que se llama `activity_main.xml`, pulsando dos veces sobre él lo abrimos para modificarlo y darle el contenido:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity" >
   <RelativeLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_centerInParent="true" >
      <ToggleButton
      android:id="@+id/btnLed0"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentLeft="true"
      android:layout_marginLeft="20dp"
      android:enabled="false"
      android:textOff="Apagado"
      android:textOn="Encendido" />
      <ToggleButton
      android:id="@+id/btnLed1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_centerInParent="true"
      android:enabled="false"
      android:textOff="Apagado"
      android:textOn="Encendido" />
      <ToggleButton
      android:id="@+id/btnLed2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentRight="true"
      android:layout_marginRight="20dp"
      android:enabled="false"
      android:textOff="Apagado"
      android:textOn="Encendido" />
   </RelativeLayout>
   <Button
      android:id="@+id/btnConnect"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignBaseline="@+id/textView1"
      android:layout_alignBottom="@+id/textView1"
      android:layout_alignLeft="@+id/textView1"
      android:text="Conectar BT" />
</RelativeLayout>
```

Todo esto nos servirá para crear la pantalla descrita anteriormente. Para introducir el código de control nos dirigimos al fichero java situado en `src/com.acme.bluetoothcontrol` y llamado `MainActivity.java`; lo abrimos del mismo modo que hicimos con el anterior y le damos el contenido:

```
package com.example.bluetoothcontrol;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Set;
import java.util.UUID;
import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.CompoundButton;
import android.widget.Toast;
import android.widget.ToggleButton;
public class MainActivity extends Activity {
   BluetoothAdapter mBluetoothAdapter;
   BluetoothSocket mSocket;
   BluetoothDevice mDevice;
   OutputStream mOutputStream;
   ToggleButton mBtn0 = null;
   ToggleButton mBtn1 = null;
   ToggleButton mBtn2 = null;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      // Botón-Led 0
      mBtn0 = (ToggleButton) findViewById(R.id.btnLed0);
      mBtn0.setOnCheckedChangeListener(new _
            CompoundButton.OnCheckedChangeListener() {
         public void onCheckedChanged(CompoundButton buttonView, _
               boolean isChecked) {
            if (isChecked) {
               sendBT(0, true);
            } else {
               sendBT(0, false);
            }
         }
      });
      // Botón-Led 1
      mBtn1 = (ToggleButton) findViewById(R.id.btnLed1);
      mBtn1.setOnCheckedChangeListener(new _
            CompoundButton.OnCheckedChangeListener() {
         public void onCheckedChanged(CompoundButton buttonView, _
               boolean isChecked) {
            // es la misma sentencia que en el boton-led 0
            sendBT(1, isChecked);
```

```
        }
    });
    // Botón-Led 2
    mBtn2 = (ToggleButton) findViewById(R.id.btnLed2);
    mBtn2.setOnCheckedChangeListener(new _
          CompoundButton.OnCheckedChangeListener() {
       public void onCheckedChanged(CompoundButton buttonView, _
             boolean isChecked) {
          sendBT(2, isChecked);
       }
    });
    // conexión/desconexión al bluetooth
    Button b = (Button) findViewById(R.id.btnConnect);
    b.setOnClickListener(new View.OnClickListener() {
       @Override
       public void onClick(View v) {
          if (mSocket != null && mSocket.isConnected()) {
             ((Button) v).setText("Conectar BT");
             try {
                endBT();
             } catch (IOException e) {
                showMessage("Error al conectar", Toast.LENGTH_SHORT);
             }
          } else {
             ((Button) v).setText("Desconectar BT");
             try {
                startBT();
             } catch (IOException e) {
                showMessage("Error al desconectar", Toast.LENGTH_SHORT);
             }
          }
       }
    });
}

// busca el dispositivo Bluetooth e inicia la conexión
void startBT() throws IOException {
    // buscar el dispositivo
    mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
    if (mBluetoothAdapter == null) {
       showMessage("No existe adaptador bluetooth disponible", _
             Toast.LENGTH_LONG);
       return;
    }
    if (!mBluetoothAdapter.isEnabled()) {
       Intent enableBluetooth = new Intent( _
             BluetoothAdapter.ACTION_REQUEST_ENABLE);
       startActivityForResult(enableBluetooth, 0);
    }
    Set<BluetoothDevice> pairedDevices = mBluetoothAdapter _
          .getBondedDevices();
    boolean deviceFound = false;
    for (BluetoothDevice device : pairedDevices) {
       if (device.getName().equals("linvor")) {  // este depende de cada
                                                 // dispositivo
```

```
                showMessage("Bluetooth disponible: " + device.getName(), _
                      Toast.LENGTH_LONG);
                mDevice = device;
                deviceFound = true;
                break;
             }
          }
          // si se encuentra abrir la conexión,
          // si no mostrar mensaje
          if (deviceFound) {
             openBT();
          } else {
             showMessage("No se ha encontrado el dispositivo indicado", _
                   Toast.LENGTH_LONG);
          }
       }
       // inicia conexión Bluetooth
       void openBT() throws IOException {
          // ID Standard
          UUID uuid = UUID.fromString("00001101-0000-1000-8000-00805f9b34fb");
          mSocket = mDevice.createRfcommSocketToServiceRecord(uuid);
          mSocket.connect();
          mOutputStream = mSocket.getOutputStream();
          enableButtons(mSocket.isConnected());
          showMessage("Bluetooth conectado", Toast.LENGTH_SHORT);
       }
       // finaliza conexión Bluetooth
       void endBT() throws IOException {
          mOutputStream.close();
          mSocket.close();
          enableButtons(mSocket.isConnected());
       }
       // muestra mensajes en pantalla
       private void showMessage(String msg, int time) {
          Toast toast = Toast.makeText(MainActivity.this, msg, time);
          toast.show();
       }
       // envío del estado deseado
       private void sendBT(int i, boolean isOn) {
          try {
             String msg = Integer.toString(i) + (isOn ? "H" : "L");
             msg += "\n";
             mOutputStream.write(msg.getBytes());
             showMessage("Datos enviados", Toast.LENGTH_SHORT);
          } catch (Exception e) {
             showMessage("Error en el envío", Toast.LENGTH_SHORT);
          }
       }
       // habilita o deshabilita los botones de los leds
       private void enableButtons(boolean isEnabled) {
          mBtn0.setEnabled(isEnabled);
          mBtn1.setEnabled(isEnabled);
          mBtn2.setEnabled(isEnabled);
       }
    }
```

Por último y antes de que iniciemos el programa, tendremos que indicar al sistema Android que vamos a usar el Bluetooth; para ello nos dirigimos al fichero `AndroidManifest.xml` y tras la línea:

```
android:versionCode="1" android:versionName="1.0" >
```

añadir la línea:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
```

Ahora ya podemos ejecutar el programa tanto en Android como en Arduino; al iniciar el programa de Android debemos pulsar en el botón para comenzar la conexión Bluetooth con Arduino y con ello, se activarán también los botones que controlarán los leds; una vez activos, al pulsar sobre ellos se irán encendiendo y apagando los leds según lo indicado.

Nota:

El hecho de necesitar una API 14 como mínimo en Android es por la llamada al método isConnected() *del objeto* BluetoothSocket, *que nos servirá para saber si existe conexión o no en el socket. En caso de tener un Android con un sistema operativo anterior a esta API, vale con eliminar estas llamadas, teniendo en cuenta que el control para saber si está conectado o no, hay que realizarlo de otra manera o tener en cuenta que podríamos enviar datos sin tener establecida la comunicación, lo que generaría un mensaje de error.*

Para controlar lo que llega a Arduino podemos aprovechar que el puerto serie estándar no lo estamos usando, añadir una líneas y volcar lo recibido en el monitor serie. Para ello deberemos configurar la conexión como siempre mediante:

```
Serial.begin(9600)
```

en la función de `setup()` y en la `loop()` añadir:

```
Serial.write(receviedChar);
```

tras obtener la variable `receviedChar`. Si abrimos el monitor serie, veremos en él una entrada por cada pulsación sobre los botones del dispositivo móvil. Véase la figura 11.6.

Con este ejemplo ya podríamos hacer que nuestra alarma se pudiera armar y desarmar a distancia o incluso controlar un juego de ordenador mediante el dispositivo móvil en lugar de usando el Joystick, pero ya que tenemos la posibilidad de tanto recibir como enviar desde Arduino, deberíamos dotar a nuestros montajes Arduino de retorno de información hacia el dispositivo

conectado; por ejemplo en caso de la alarma, saber qué estado se encuentra, si esta armada o no, o en el del juego mostrar el combustible que le queda a la nave o la munición. Para ver cómo se devuelve información hacia el dispositivo conectado realizaremos un chat por Bluetooth con un dispositivo Android.

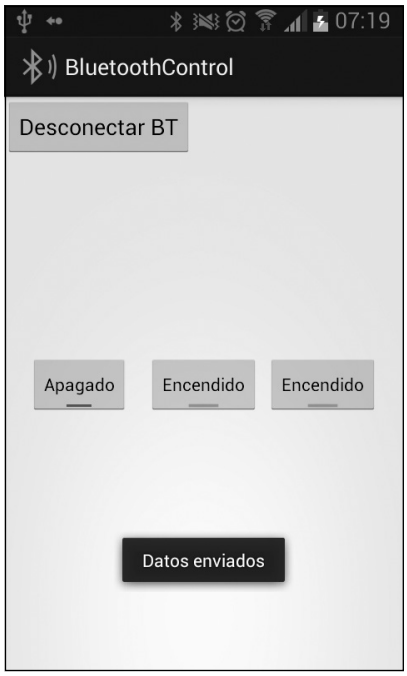


Figura 11.6. Aplicación Android en funcionamiento.

Chat Bluetooth

Si existe una aplicación en la que está claro que existe información tanto enviada como recibida, esa es un chat. Para comprobar la facilidad con la que podemos tanto enviar como recibir datos desde un circuito Arduino con Bluetooth, realizaremos un chat en el que se conversará con cualquier dispositivo que se le conecte; en nuestro caso será nuevamente con un Android, pero es fácilmente extrapolable a otros sistemas operativos y dispositivos. Para el envío de los datos por parte de Arduino, utilizaremos nuevamente el monitor serie, con lo que las conexiones del módulo Bluetooth las realizaremos sobre los pines 2 y 3 como en el ejemplo anterior; si se dispone de

un teclado para conectar a Arduino, el programa es fácilmente modificable para poder utilizarlo. Para este circuito simplemente necesitaremos el módulo Bluetooth y como acompañamiento el dispositivo Android.

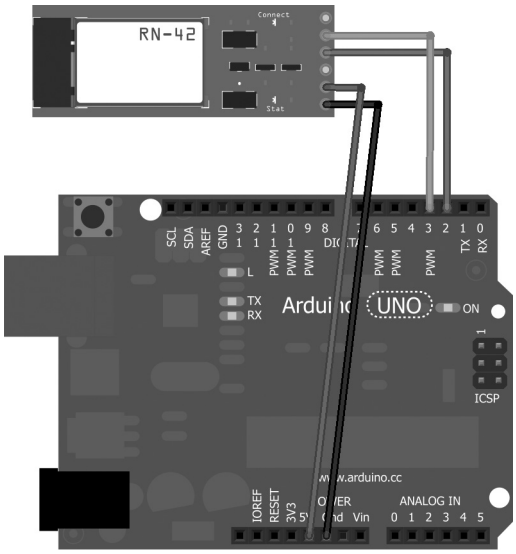


Figura 11.7. Circuito para chat Bluetooth.

El *sketch* será muy parecido al anteriormente realizado, pero en esta ocasión no sólo leeremos los datos recibidos por el Bluetooth sino también se debe enviar lo que el usuario introduzca por el monitor serie, por lo que deberemos leer la entrada serie y en caso de obtener datos en ella, se deben enviar.

```
#include <SoftwareSerial.h>

int bluetoothTx = 2;
int bluetoothRx = 3;

SoftwareSerial bluetooth(bluetoothTx, bluetoothRx);  // instancia serie para
                                                     // Bluetooth

void setup(){
   // Conexión con el ordenador
   Serial.begin(9600);

   // Realizamos la conexión Bluetooth con el dispositivo móvil
   bluetooth.begin(9600);
}

void loop(){
   // obtenemos lectura de Bluetooth
   if(bluetooth.available()){
```

```
      char toSend = (char)bluetooth.read();
      // escribimos en el monitor
      Serial.print(toSend);
   }

   // obtenemos lectura del monitor
   if(Serial.available()){
      // mientras haya datos, enviar
      while (Serial.available()){
         char toSend = (char)Serial.read();
         // escribimos en el Bluetooth
         bluetooth.print(toSend);
      }
      // fin de transmisisón
      bluetooth.println();
   }
}
```

Como podemos ver en este caso también es necesario apoyarnos en la clase `SoftwareSerial` dado que necesitamos dos transmisiones serie y no utilizamos los pines de puerto serie estándar para la transmisión Bluetooth.
En cuanto a la lógica de la función `loop()`, se realiza la lectura del puerto `bluetooth` y lo que se va leyendo de él, se va volcando en el monitor serie; tras cada lectura de un byte de entrada de Bluetooth, se realiza una comprobación de si hay algo pendiente de enviar, pero en este caso en lugar de enviar un byte y volver a leer si hay algo pendiente en la entrada `bluetooth`, lo que haremos es leer todo lo que haya pendiente de enviar y enviarlo, así nos será más fácil procesarlo en Android, es decir, enviamos frases completas y evitamos quedarnos a media comunicación. Para ello se mira si hay algo pendiente en el puerto `Serial` y en caso de haberlo se produce el bucle `while()` que irá sacando los bytes del `buffer` y enviándolos hacia el puerto `bluetooth`. Para acabar la transmisión y dar por finalizado el mensaje actual enviamos una señal de nueva línea (código ASCII 10), cosa que hacemos mediante:

```
bluetooth.println();
```

La parte correspondiente a Arduino ya está finalizada, vayamos ahora a la parte correspondiente a Android.
Generamos un nuevo proyecto de al misma forma que lo creamos para el ejemplo anterior, pero esta vez con los parámetros:

- Application Name: BluetoothChat.
- Project Name: BluetoothChat.
- Package Name: com.acme.bluetoothchat.
- Minimum Required SDK: API 14.

Nuevamente miramos que la opción **BlankActivity** en el paso correspondiente a la creación de actividades esté seleccionada por defecto.

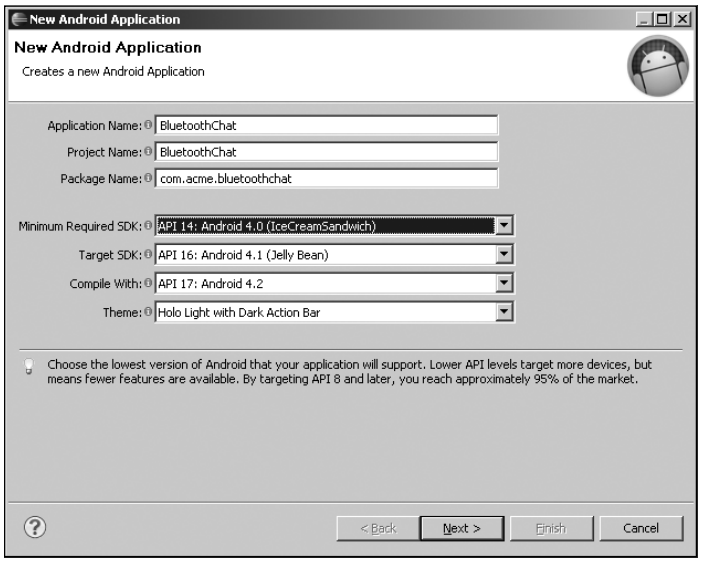


Figura 11.8. Configuración del proyecto Android.

En el archivo `activity_main.xml` situado en la carpeta `res/layout`, que como recordaremos era para definir el aspecto gráfico, introduciremos el código correspondiente a una pantalla con un botón para la conexión y desconexión del Bluetooth, un texto donde se irá guardando la conversación mantenida y una caja de texto con un botón para poder enviar mensajes hacia Arduino. El código para obtener la pantalla es:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent" >

   <Button
   android:id="@+id/open_close"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_alignParentTop="true"
   android:layout_centerHorizontal="true"
   android:text="Conectar BT" />

   <RelativeLayout
   android:id="@+id/main_control"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
```

```
    android:layout_alignParentBottom="true"
    android:layout_margin="5dip" >

    <TextView
    android:id="@+id/label"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true"
    android:text="Mensaje a enviar" />

    <EditText
    android:id="@+id/entry"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/label"
    android:background="@android:drawable/editbox_background"
    android:enabled="false" />

    <Button
    android:id="@+id/send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/entry"
    android:enabled="false"
    android:text="Enviar" />
</RelativeLayout>

<TextView
android:id="@+id/receivedData"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:layout_above="@+id/main_control"
android:layout_alignParentLeft="true"
android:layout_below="@+id/open_close"
android:layout_margin="5dip"
android:background="#bbbbbb"
android:gravity="bottom"
android:maxLines="100" >
</TextView>

</RelativeLayout>
```

En el archivo de código java llamado `MainActivity.java` situado en `src/com.acme.bluetoothchat`, introduciremos la lógica del programa para controlar las comunicaciones con Arduino mediante Bluetooth; el inicio de la conexión y desconexión es prácticamente igual al ejercicio anterior. El envío de los datos también es muy semejante, pero en este caso el lugar de obtener el estado de pulsación de un botón se recupera el texto introducido por el usuario y será lo que se envíe. Por último está la parte de recepción, que se trata de un *thread* que va leyendo la entrada del *socket* a través de un objeto

de clase `InputStream`; cuando hay datos para leer, los va recuperando hasta encontrar un carácter de nueva línea (recordemos que en Arduino cada vez que enviábamos datos, acabábamos enviando el carácter ASCII 10 marcando nueva línea y por lo tanto fin de mensaje actual), momento en el cual detecta el fin del mensaje y crea una cadena de texto que muestra en pantalla.
El código de este fichero es:

```
package com.example.bluetoothchat;
package com.example.bluetoothchat;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Set;
import java.util.UUID;

import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends Activity {

   EditText mTextToSend;
   TextView mReceivedData;
   Button mSend;

   BluetoothAdapter mBluetoothAdapter;
   BluetoothSocket mSocket;
   BluetoothDevice mDevice;
   OutputStream mOutputStream;
   InputStream mInputStream;
   Thread workerThread;
   byte[] readBuffer;
   int readBufferPosition;
   int counter;
   volatile boolean stopWorker;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      // botón de control de comunicación
      findViewById(R.id.open_close).setOnClickListener( _
            new View.OnClickListener() {
```

```
            public void onClick(View v) {
               if (mSocket != null && mSocket.isConnected()) {
                  ((Button) v).setText("Conectar BT");
                  try {
                     endBT();
                  } catch (IOException e) {
                     e.printStackTrace();
                     showMessage("Error al conectar", Toast.LENGTH_SHORT);
                  }
               } else {
                  ((Button) v).setText("Desconectar BT");
                  try {
                     startBT();
                  } catch (IOException e) {
                     showMessage("Error al desconectar", Toast.LENGTH_SHORT);
                  }
               }
            }
         });

         // Botón de envío
         mSend = (Button) findViewById(R.id.send);
         mSend.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
               sendBT();
            }
         });

         mTextToSend = (EditText) findViewById(R.id.entry);
         mReceivedData = (TextView) findViewById(R.id.receivedData);

      }

      // busca el dispositivo Bluetooth e inicia la conexión
      void startBT() throws IOException {
         // buscar el dispositivo
         mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

         if (mBluetoothAdapter == null) {
            showMessage("No existe adaptador bluetooth disponible", _
                  Toast.LENGTH_LONG);
            return;
         }

         if (!mBluetoothAdapter.isEnabled()) {
            Intent enableBluetooth = new Intent(
            BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBluetooth, 0);
         }

         Set<BluetoothDevice> pairedDevices = mBluetoothAdapter
            .getBondedDevices();

         boolean deviceFound = false;
         for (BluetoothDevice device : pairedDevices) {
```

```
      if (device.getName().equals("linvor")) {  // este depende de cada
                                                // dispositivo
         showMessage("Bluetooth disponible: " + device.getName(), _
               Toast.LENGTH_LONG);
         mDevice = device;
         deviceFound = true;
         break;
      }
   }
   // si se encuentra abrir la conexión, si no mostrar mensaje
   if (deviceFound) {
      openBT();
   } else {
      showMessage("No se ha encontrado el dispositivo indicado", _
            Toast.LENGTH_LONG);
   }
}

// muestra mensajes en pantalla
private void showMessage(String msg, int time) {
   Toast toast = Toast.makeText(MainActivity.this, msg, time); _
         toast.show();
}

// inicia conexión Bluetooth
void openBT() throws IOException {
   // ID Standard
   UUID uuid = UUID.fromString("00001101-0000-1000-8000-00805f9b34fb");
   mSocket = mDevice.createRfcommSocketToServiceRecord(uuid);
   mSocket.connect();
   mOutputStream = mSocket.getOutputStream();
   enableControls(mSocket.isConnected());
   mInputStream = mSocket.getInputStream();
   startDataListener();
   showMessage("Bluetooth conectado", Toast.LENGTH_SHORT);
}

// habilita o deshabilita los controles de envío
private void enableControls(boolean isEnabled) {
   mSend.setEnabled(isEnabled);
   mTextToSend.setEnabled(isEnabled);
}

void startDataListener() {
   final Handler handler = new Handler();
   final byte delimiter = 10; // control de nueva línea ASCII 10

   stopWorker = false;
   readBufferPosition = 0;
   readBuffer = new byte[1024];
   workerThread = new Thread(new Runnable() {
   public void run() {
   while (!Thread.currentThread().isInterrupted() && !stopWorker) {
      try {
         int bytesAvailable = mInputStream.available();
```

```
            if (bytesAvailable > 0) {
               byte[] packetBytes = new byte[bytesAvailable];
               mInputStream.read(packetBytes);
               for (int i = 0; i < bytesAvailable; i++) {
                  byte b = packetBytes[i];
                     if (b == delimiter) {
                        byte[] encodedBytes = new byte[readBufferPosition];
                        System.arraycopy(readBuffer, 0, _
                              encodedBytes, 0, encodedBytes.length);
                        final String data = new String( _
                              encodedBytes, "US-ASCII");
                        readBufferPosition = 0;

                        handler.post(new Runnable() {
                        public void run() {
                     mReceivedData.setText(mReceivedData.getText().toString()
                     + "\n"
                     + "Arduino dice: "
                     + data);
                     }
                     });
                     } else {
                        readBuffer[readBufferPosition++] = b;
                     }
                  }
               }
            } catch (IOException ex) {
               stopWorker = true;
            }
         } // while
         }
         }); // thread

         workerThread.start();
      }

      // envío del mensaje
      void sendBT() {
         try {
            String msg = mTextToSend.getText().toString();
            msg += "\n";
            mOutputStream.write(msg.getBytes());
            mReceivedData.setText(mReceivedData.getText().toString() + "\n" _
                  + "Android dice: " + msg);
            // mostramos en el log de conversación
            showMessage("Datos enviados", Toast.LENGTH_SHORT);
            // limpiamos la caja
            mTextToSend.setText("");
         } catch (Exception e) {
            showMessage("Error en el envío", Toast.LENGTH_SHORT);
         }
      }

      // finaliza conexión Bluetooth
      void endBT() throws IOException {
         stopWorker = true;
```

```
        mOutputStream.close();
        mInputStream.close();
        mInputStream.close();
        mSocket.close();
    }
}
```

Para poder trabajar con Bluetooth en Android es necesario dar permisos a la aplicación. Esto se realiza a través de su fichero `AndroidManifest.xml`. Debemos modificarlo y tras la línea:

```
android:versionCode="1" android:versionName="1.0" >
```

añadimos la línea:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
```

Ahora ya podemos ejecutar el programa y disfrutar de nuestro chat Bluetooth.

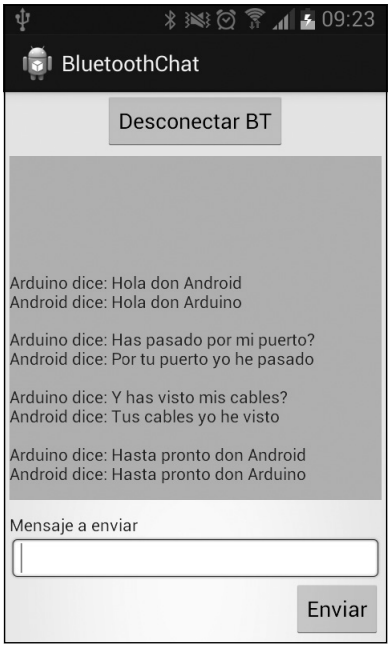


Figura 11.9. Aplicación chat Android en funcionamiento.

Nota:

*Puede obtener más información sobre cómo realizar aplicaciones Android en el libro "*Desarrollo de Aplicaciones para Android*" de esta misma colección y autor.*





12 Internet

En este capítulo aprenderá a:

- Conocer los tipos de conexiones disponibles.
- Utilizar las librerías disponibles para cada una de ellas.
- Crear conexiones y consumir servicios de Internet.
- Hacer disponibles los datos de Arduino a través de Internet.

Mediante Arduino podemos comunicarnos con infinidad de dispositivos; ya se han visto algunos tipos de comunicaciones como las realizadas con el PC mediante conexión serie, con dispositivos móviles por Bluetooth o bien con el televisor mediante infrarrojos, pero no nos vamos a quedar aquí puesto que podemos comunicarnos con Internet entero.

Para las comunicaciones con Internet desde Arduino, podemos valernos de distintos tipos de conexiones; quizá la más sencilla de todas sea mediante cable RJ45, con el que nos conectaremos nuestra red y desde ahí se accederá a Internet pero estamos atados a no poder mover mucho nuestro circuito por impedimentos del cable; otra opción es mediante la conexión a una red WiFi que nos da la posibilidad de no depender del cable y tener mayor movilidad, pero siempre se debe vigilar no salirse del radio de alcance de la cobertura de algún punto de acceso de la WiFi; en caso de que se necesite movilidad absoluta, se optará por conexiones telefónicas en alguna de sus variantes.

No existe ninguna *shield* que pueda utilizarse de las tres formas, pero si podemos encontrar algunas compuestas que ofrezcan las conexiones de cable y WiFi o bien otras con WiFi y conexión telefónica; no obstante lo más normal es que las *shield* sean dedicadas al tipo de conexión que se va a utilizar.

Para el uso de estas conexiones tenemos disponibles de manera estándar unas librerías para cada caso. La librería base podríamos decir que es la Ethernet y tanto la WiFi como la de conexión telefónica se basan en ella y comparten muchas funciones de lo que sería el núcleo de operaciones a realizar. Actualmente de manera estándar sólo se soportan conexiones GSM (de hecho la librería se llama GSM), aunque algunos fabricantes tienen *shields* que soportan conexiones 3G y 4G con sus propias librerías.

Puesto que tanto la librería GSM como la librería WiFi son variaciones de la librería Ethernet y las llamadas son casi idénticas, veremos más en profundidad las conexiones Ethernet, pero si el lector dispone de una *shield* GSM o WiFi, los ejemplos que se verán a continuación son igualmente válidos salvo pequeños detalles.

Ethernet

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 y el otro se encarga de consumirlo. La librería Ethernet dispone de un conjunto de clases que nos permitirán conectarnos a Internet y actuar desde nuestra tarjeta Arduino tanto en modo cliente como en modo servidor. Internamente

la librería Ethernet utiliza la comunicación con la *shield* mediante SPI (Serial Peripheral Interface, Interfaz de Periféricos Serie), por lo que muchas de las llamadas a funciones serán muy parecidas a las ya utilizadas.

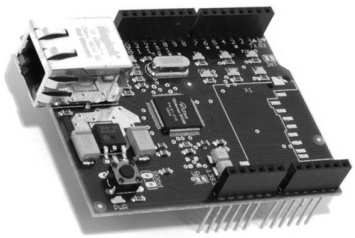


Figura 12.1. *Shield* Ethernet.

Para comenzar a trabajar con esta librería, se debe incorporar al código por medio de la inclusión de las líneas:

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

O bien seleccionando el menú Sketch>Importar librería...>Ethernet y luego Sketch>Importar librería...>SPI.
Veremos que al incluir la librería Ethernet desde menú, se incluyen muchas más líneas en la parte superior del *sketch*; podemos borrar las que sobran.
Una vez incluida la librería, se debe iniciar la *shield* Ethernet mediante la llamada a:

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

El primer parámetro es el único obligatorio y se trata de la MAC (*Media Access Control*) que es la identificación del hardware; esta identificación se supone única para cada dispositivo que se pueda conectar a una red. En muchos dispositivos viene el número de MAC en una pegatina o con serigrafía, pero si no se tiene se puede poner cualquiera siempre que no coincida con la MAC de alguno de los dispositivos que tengamos en la red; si se usa un firewall o DHCP por MAC también habrá que tenerlo en cuenta a la hora de informar el parámetro. En Arduino, la representación de la MAC viene dada por un array de seis bytes. Si no se informa ningún parámetro más, obtendremos la dirección IP mediante DHCP, es decir automáticamente. Si se informa el parámetro `ip`, podremos indicar la dirección IP que se quiere que tenga el dispositivo dentro de la red (debe cumplir las especificaciones de la red o no funcionará correctamente); viene dado por un array de 4 bytes. El parámetro `dns` permite informar la IP del servidor de nombres a utilizar; se informa mediante un array de 4 bytes. El parámetro `gateway` sirve para informar el

gateway de salida; nuevamente se trata de una IP que se informa mediante un array de 4 bytes; si no se informa se utilizará como gateway la misma IP que se tiene en el dispositivo pero con el último octeto a 1, por ejemplo 192.168.1.78 usaría el gateway 192.168.1.1. Por último el parámetro `subred` nos sirve para indicar la máscara de red a utilizar; por defecto se usa la 255.255.255.0. Un ejemplo de inicialización sería:

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

byte mac[] = { 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF };

void setup(){
   Ethernet.begin(mac);
}

void loop () {}
```

Si esta llamada se utiliza sólo con el parámetro MAC (es decir que se obtiene la IP mediante DHCP), devuelve 1 si se ha recibido IP de modo correcto, el resto de las formas de inicialización no devuelven nada.
Usar DHCP tiene muchas ventajas, como que no tenemos que preocuparnos de las complejidades de la topología de la red, pero también tiene algún inconveniente, por ejemplo, en caso de que queramos utilizar nuestro Arduino como servidor, necesitamos saber la IP, para ello podemos recuperarla en cualquier momento, por ejemplo para imprimir la IP actual usaríamos:

```
Serial.println(Ethernet.localIP());
```

Otro problema que tenemos como servidor, es que cuando se obtiene una IP por DHCP, se obtiene por un tiempo determinado y luego caduca. Para poder mantener la IP hace falta renovar este tiempo de caducidad; para mantener la IP está la función:

```
Ethernet.maintain();
```

Que devolverá un byte con el resultado de su ejecución:

Valor del byte	Resultado
0	No ha pasado nada.
1	Renovación fallida.
2	Renovación exitosa.
3	Error al enlazar.
4	Enlace exitoso.

Una vez que tenemos configurada la IP de nuestra tarjeta ya podemos actuar tanto en el rol de cliente como en el rol de servidor de la conexión. Pasemos a ver unos ejemplos sobre cómo funciona.

Cliente

Cuando la tarjeta Arduino se conecta a alguna dirección de Internet (o de nuestra red privada) para obtener datos, diremos que estamos trabajando en modo cliente.
Para comenzar a trabajar en modo cliente necesitamos una instancia de la clase `EthernetClient`, que nos dará la opción de realizar las conexiones. La instancia se obtiene mediante:

```
EthernetClient client;
```

Una vez obtenida la instancia la podemos usar para conectarnos a los servidores a través de la función:

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

teniendo como parámetros `servidor` que es el servidor al que queremos conectarnos y viene dado por su IP o por su nombre de dominio y `puerto` que es el puerto al que se tiene que conectar para obtener el servicio por parte del servidor.
Una vez conectados al servidor, podemos comunicarnos con él a través de unas funciones ampliamente utilizadas en ejemplos anteriores, se tratan de las funciones `write()`, `print()`, `println()`, `available()` y `read()` que funcionan de la misma manera a la ya vista, solamente que aquí aplica a los datos a transmitir o recibir del servidor.
Para saber si nos encontramos conectados, podemos consultar a la función `client.connected()` que devolverá `true` en caso de estarlo y `false` si no lo estamos.
Para ilustrar el funcionamiento de estas funciones haremos un cliente HTTP muy sencillo; nos conectaremos a un servicio en internet para obtener 10 números aleatorios del 1 al 100 en base 16; la dirección que escribiríamos en el navegador para conectarnos al servicio sería:

```
http://www.random.org/integers/?num=10&min=1&max=100&col=1&base=16&format=
plain&rnd=new.
```

En el *sketch* debemos conectarnos a este servidor cuya IP es 67.23.25.127 y una vez conectados al servicio HTTP realizar un GET de la cadena "`/integers/?num=10&min=1&max=100&col=1&base=16&format=plain&rnd=new`".

Tras realizarse la conexión el servidor habrá devuelto la respuesta, momento en el cual la llamada `client.available()` devolverá `true` y podremos comenzar a leer los datos transmitidos mediante `client.read()`.

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

byte mac[] = { 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF };
IPAddress server(67,23,25,127);

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 conexion...");
   if (client.connect(server, 80)) {
      Serial.println("Conectados!");
      // Petición HTTP
      client.println("GET /integers/?num=10&min=1&max=100&col=1&base=
16&format=plain&rnd=new HTTP/1.0");
      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(;;){}
   }
}
```

A la hora de realizar la petición GET, se envía la cadena de petición y acto seguido se invoca una escritura con retorno de carro pero sin argumentos:

```
client.println();
```

Esto es porque en las conexiones HTTP, se debe terminar cada petición con una línea en blanco. Tal como podemos ver, la lectura de los datos una vez enviada la petición GET, es muy semejante a como se hacía en las comunicaciones serie y es que estamos usando SPI.
Dentro del listado podemos encontrar una forma del `for()` que hasta ahora no habíamos utilizado:

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

Con esta estructura de instrucción, se consigue un bucle infinito, nunca saldremos de esa instrucción; se ha puesto una vez que estemos desconectados del servidor, ya que no volveremos a recibir más datos de éste y es una manera de bloquear la ejecución.
Como vemos puede ser muy interesante que la tarjeta actúe como cliente, por ejemplo podemos obtener un RSS de noticias o algún servicio que nos dé el tiempo local y dependiendo de si va a hacer sol activar el motor de un toldo que tape nuestra galería o si va a llover parar el riego a aspersión, todo sin tener que estar presentes.

Servidor

Si la opción de trabajar como cliente de un servicio era interesante, quizá encuentre el lector más interesante la opción servidor. Al estar como servidor, ofreceremos un puerto de comunicaciones al cual un cliente se podrá conectar para interactuar con la tarjeta o recibir información; ¿qué tipo de información? pues la que consideremos en cada caso.
Para trabajar como servidor existe la clase `EthernetServer` que nos permitirá generar un servidor para que escuche sobre el puerto indicado.

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

Una vez creado el servidor se indica que debe comenzar a escuchar por el puerto dado mediante la instrucción `server.begin()`. Desde este momento nuestro servidor es capaz de atender peticiones de clientes (hasta cuatro de manera concurrente).
Para detectar que hay un cliente esperando a ser atendido usaremos la función siguiente:

```
server.available()
```

que devuelve un objeto de tipo `EthernetClient` en el caso de que exista un cliente pendiente o un objeto nulo si no hay ningún cliente, por lo que podemos evaluarlo en una estructura `if()`. El objeto cliente obtenido puede ser consultado para leer los datos de la petición, por ejemplo para mostrar en el monitor serie lo enviado por el cliente:

```
EthernetClient client = server.available();
if (client) {
   while (client.connected()) {
      if (client.available()) {
         Serial.write(client.read());
      }
   }
}
```

También podemos enviar datos a los clientes conectados usando las funciones:

```
server.write(data);
server.print(data);
server.println(data);
```

que trabajan de modo semejante a las vistas anteriormente, es decir, la función `write()` para escribir a nivel de byte y las funciones `print()` y `println()` para hacerlo en ASCII.

Para ver cómo funciona el servidor, vamos a partir de uno de los ejemplos que acompañan al entorno de programación; se trata de un pequeño un servidor HTTP de modo que desde nuestro navegador podemos conectarnos a él para ver los valores de las entradas de nuestra tarjeta. Para abrir el ejemplo pulsamos el menú Archivo>Ejemplos>Ethernet>WebServer. A continuación veremos las partes más interesantes del código obtenido.

Después de la inclusión de los archivos de cabecera, tenemos la definición de la dirección MAC a utilizar y la IP que usará nuestro servidor. En el ejemplo anterior habíamos obtenido la IP mediante DHCP y podríamos haber usado esa misma técnica aquí y recuperar la IP asignada mediante `Ethernet.localIP()`, pero en este caso se realizará una asignación manual, para lo cual se genera una instancia de la clase IPAdress con la dirección deseada y que se utilizará más adelante.

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

Nota:

Dependiendo de la configuración de red que se disponga en cada caso, es posible que sea necesario cambiar la dirección IP asignada. En caso de duda, se puede utilizar DHCP y obtener la IP asignada por medio del monitor serie.

El servidor se define para que trabaje sobre el puerto 80, que es el puerto habitual del protocolo HTTP utilizado por los navegadores.

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

En la función `setup()` además de preparar el monitor serie, también se inicia la Ethernet con la MAC y la IP especificadas y se comienzan las escuchas por parte del servidor al puerto configurado.

```
Ethernet.begin(mac, ip);
server.begin();
```

Dentro de la función `loop()` es donde se esperan los clientes del servicio ofrecido y en cada iteración del bucle se mira si hay algún cliente a la espera de ser atendido mediante:

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

En caso de existir algún cliente, se mira si tiene algún dato para ser leído de su petición.

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

Además se guarda una variable llamada `currentLineIsBlank` para conocer si es final de petición o no.
De modo general su funcionamiento es éste: 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), o en otras palabras 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) {
```

Aquí la respuesta además de los datos obtenidos en la tarjeta debe devolver una cabecera estándar HTTP para poder ser interpretada por el navegador:

```
client.println("HTTP/1.1 200 OK");
client.println("Content-Type: text/html");
   ...
```

Los datos leídos en la tarjeta Arduino, se deben devolver formateados en HTML con tal de que el navegador pueda mostrar la información de manera correcta. El programa recorre los seis canales analógicos realizando lecturas y mostrándolas una por cada línea.

```
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;
```

La parte del código correspondiente a la etiqueta `meta` permitirá que la página Web se vaya recargando automáticamente cada 5 segundos sin necesidad de interactuar nosotros.
Finalmente, una vez enviada la página Web, se cierra la conexión con el cliente a la espera de una nueva conexión por parte de éste.

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

Si ejecutamos el *sketch* y nos conectamos desde el navegador a la dirección IP configurada, obtendremos una página con los valores de las entradas analógicas en ese momento.

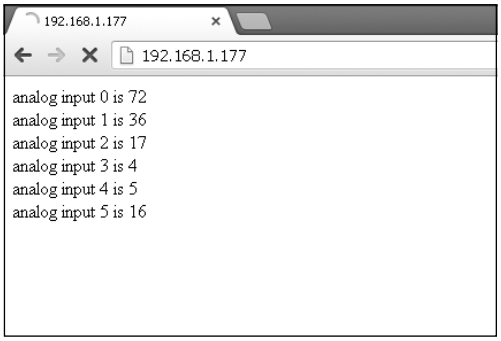


Figura 12.2. Resultado del programa de ejemplo en el navegador.

Esta salida tan espartana, para el uso particular puede estar bien, pero para nosotros no es suficiente, asi que vamos a mejorarla un poco.

Servidor Web de temperatura y humedad

Ya hemos trabajando anteriormente con el sensor de temperatura y humedad DHT21 por lo que no esconderá ningún secreto la forma de obtener sus lecturas (ver capítulo 8 en caso de dudas).

Para la conexión del circuito se debe tener en cuenta que la *shield* de Ethernet será quien esté directamente pinchada a la placa Arduino, y que será sobre la *shield* sobre la que se tengan que hacer las conexiones; no obstante, normalmente las *shield* traen también con serigrafía los números correspondientes a los pines Arduino.

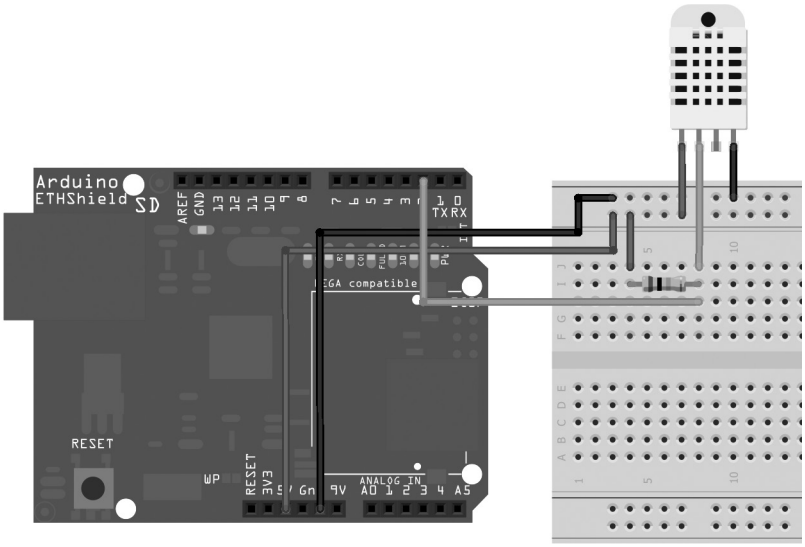


Figura 12.3. Circuito de lectura de temperatura por Internet.

Lo que vamos a hacer es leer como se hizo en el capítulo 8 los valores del sensor, pero los incluiremos en la página Web a mostrar. Guardaremos un array con los valores leídos, de modo que la página muestre siempre los últimos 10 valores obtenido, pero en lugar de hacerlo en modo texto como en el ejemplo anterior, aprovecharemos una de las API que Google pone a disposición de los usuarios y generaremos un gráfico con las temperaturas y las humedades recogidas.

En la función `setup()` configuramos tanto el pin de entrada del sensor como el servidor Web, tal y como se hizo en el ejemplo anterior. En la función `loop()` leeremos el sensor cada segundo ya que necesita cierto tiempo para realizar la lectura completa y si realizamos lecturas con una mayor frecuencia podemos incurrir errores.

Para no bloquear la ejecución usando un `delay()` y que durante ese segundo el servidor Web dejara de atender peticiones, lo que haremos será usar una técnica ya conocida que es guardar el tiempo en el que se hace la lectura y calcular una diferencia con el tiempo actual de 1000 ms:

```
if ((millis() - lastTime) > 1000){
   lastTime = millis();
   readSensor();
}
```

Cuando generemos la página Web, existirán dos partes; la primera de ellas se configura lo que se quiere mostrar, un array en javascript con los valores leídos, y una segunda parte donde se le indica a la página que muestre dicho array. Se debe tener en cuenta que el sensor podía dar errores durante la lectura de los valores, en tal caso, la variable `errorCode` contendrá un valor distinto de 0, con lo que podemos seleccionar si deseamos mostrar un error o los valores recogidos. Si el valor es distinto de 0, entonces mostraremos error y en la página Web no hará falta generar el array con los valores.

```
if (errorCode == 0){
   // preparar array con los valores a mostrar
}
...
switch (errorCode){
   case 0: // no hay errores mostramos la salida
      // mostrar array
      break;
   // Errores
   case 1:
      // mostrar error 1
      break;
...
}
```

En cuanto a los valores recogidos en el *sketch*, tendremos que procesarlos antes de guardar. Como vimos en el capítulo 8, en este sensor vienen por separado las partes enteras de las decimales, tanto de la temperatura como de la humedad y vienen como tipos enteros. Nosotros lo guardaremos en un array de tipo `float` para ocupar menor espacio y ser más fáciles de manejar, por lo que tendremos que realizar alguna operación antes de guardarlos.
La operación consiste en transformar los valores leidos en cadena de texto donde se juntarán la parte entera y la decimal separadas por un punto; una vez está formada esta cadena de texto, ya podemos pasar de modo sencillo a tipo `float`:

```
char buf[10];
snprintf(buf, sizeof buf, "%d.%d", dhtData[0], dhtData[1]);
values[index][0] = atof(buf);
```

Dentro del array `values`, se guardará la humedad en la primera posición y la temperatura en la segunda. El array `values` será de tamaño igual a la constante definida `MAX_READS`, y tendremos un índice que se irá manteniendo de manera que apunte a la siguiente posición libre donde se pueda guardar

el valor. El problema es que cuando el índice que apunta al array llegue a la posición MAX_READS, significará que no quedan más espacios que rellenar en el array; en este caso lo que haremos será desplazar todos los valores a una posicón menos dentro del array de modo que quede libre el índice MAX_READS - 1 y podamos escribir en el; los valores que se guardaran en ese momento en la posición 0 se perderían, pero de este modo siempre tendremos en el array los últimos MAX_READS valores más recientes.

```
void shiftArray(){
   for (byte i = 1; i <MAX_READS; i++){
      values[i-1][0] = values[i][0];
      values[i-1][1] = values[i][1];
   }
}
```

El código completo quedaría:

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

// número de lecturas a guardar
const byte  MAX_READS = 10;
const byte dhtPin = 2; // pin del sensor
byte errorCode; // código de error
byte dhtData[5]; // bytes leídos
// array con la información de temperaturas y humedad
float values[MAX_READS][2];
int index = 0; // índice para el array
long lastTime=0; // tiempo de lectura

byte mac[] = { 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF };
IPAddress ip(192,168,1, 177);

// Configuramos el servicio en el puerto 80
EthernetServer server(80);

void setup() {
   Serial.begin(9600);
   pinMode(dhtPin,OUTPUT);
   digitalWrite(dhtPin,HIGH);
   // Iniciamos las conexiones
   Ethernet.begin(mac, ip);
   // Inciamos e servidor
   server.begin();
   Serial.print("Servidor disponible en la IP: ");
   Serial.println(Ethernet.localIP());
}

void loop() {
   if ((millis() - lastTime) > 1000){
      lastTime = millis();
      readSensor();
   }
```

```
// Escucha de clientes
EthernetClient client = server.available();
if (client) {
   Serial.println("Se detecta petición");
   // controla el fin de petición HTTP
   boolean currentLineIsBlank = true;
   while (client.connected()) {
      if (client.available()) {
         char c = client.read();
         Serial.write(c);
         // Si se recibe un final de línea y la línea está
         // en blanco indica el final de petición
         // si se usan parámetros es el momento de procesar
         // el buffer donde se guarden los caracteres
         // Podemos comenzar a responder
         if (c == '\n' && currentLineIsBlank) {
            // Se debe enviar la cabecera estándar HTTP
            client.println("HTTP/1.1 200 OK");
            client.println("Content-Type: text/html");
            client.println("Connection: close");
            client.println();
            client.println("<!DOCTYPE HTML>");
            client.println("<html>");
            client.println("<meta http-equiv=\"refresh\" content=\"5\">");
            client.println("<head>");
            if (errorCode == 0){
               client.println("<script type=\"text/javascript\" _
                     src=\"https://www.google.com/jsapi\"></script>");
               client.println("<script type=\"text/javascript\">");
               client.println("google.load(\"visualization\", \"1\", _
                     packages:[\"corechart\"]});");
            client.println("google.setOnLoadCallback(drawChart);");
               client.println("function drawChart() {");
               client.println("var data = google.visualization._
                     arrayToDataTable([");
               client.println("['Toma', 'Humedad', 'Temperatura'],");
               for (byte i = 0 ; i < index ; i ++){
                  client.print("['");
                  client.print(i);
                  client.print("',");
                  client.print(values[i][0]);
                  client.print(",");

                  client.print(values[i][1]);
                  client.print("]");
                  if (i<index-1){
                     client.print(",");
                  }
                  else{
                     client.print("]);");
                  }
               }

               client.println("var options = {title: _
                     'Humedad y Temperatura',");
```

```
            client.println("hAxis: {title: 'Lecturas', _
                  titleTextStyle: {color: 'red'}}}; ");

            client.println("var chart = new google.visualization._
                  AreaChart(document.getElementById('chart_div'));");
            client.println("chart.draw(data, options);");
            client.println("}");
            client.println("</script>");
         }

         client.println("</head>");
         client.println("<body>");
         switch (errorCode){
            case 0: // no hay errores mostramos la salida
               client.println("<div id=\"chart_div\" style=\"width: _
                     900px; height: 500px;\"></div>");
               break;
            //Errores
            case 1:
               client.println("Error 1: Ha fallado la primera _
                     condicion de comienzo.");
               break;

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

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

            default:
               client.println("Error no conocido.");
               break;
         }
         client.println("</body>");
         client.println("</html>");
         break;
      }
      if (c == '\n') {
         // se comienza nueva línea
         currentLineIsBlank = true;
      }
      else if (c != '\r') {
         // tenemos un caracter dentro de esta línea, así pues no
         // está vacío
         currentLineIsBlank = false;
      }
      // si se usan parámetros, se guardaría el caracter para
      // procesarlo más adelante.
   }
}
// tiempo de retardo de seguridad
delay(100);
```

```
      // se cierra la conexión
      client.stop();
      Serial.println("Desconexión de cliente");
   }
}

void shiftArray(){
   for (byte i = 1; i <MAX_READS; i++){
      values[i-1][0] = values[i][0];
      values[i-1][1] = values[i][1];
   }
}

void readSensor(){
   readDHT(); // comenzamos la lectura
   if (errorCode == 0){
      if (index == (MAX_READS)){
         index--;
         shiftArray();
      }

      char buf[10];
      snprintf(buf, sizeof buf, "%d.%d", dhtData[0], dhtData[1]);
      values[index][0] = atof(buf);
      snprintf(buf, sizeof buf, "%d.%d", dhtData[2], dhtData[3]);
      values[index][1] = atof(buf);
      // ajustamos índice
      index++;
   }

}

/*****************************************************
* 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;
}
```

En el gráfico obtenido aparece en el eje de abcisas el número de posición del valor en el array, de modo que cuando éste está completamente informado, aparecerán los números de 0 a `MAX_READS-1`. Si se desea que en lugar de estos números aparezca el momento en el que se ha producido la lectura (día y hora por ejemplo), se puede utilizar la librería Time y obtener estos valores.

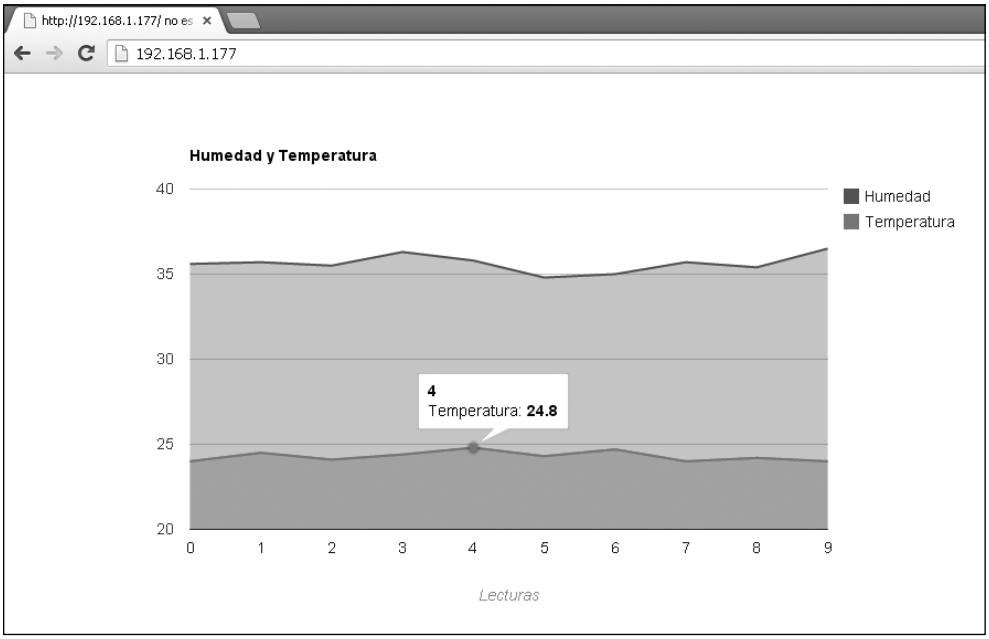


Figura 12.4. Gráfico obtenido en la de temperatura por Internet.

WiFi

Como hemos dicho, la librería WiFi se basa en la librería Ethernet, pero tiene algunas peculiaridades porque también las tiene la conexión en sí.
Por ejemplo, para poderse conectar a una red Ethernet, realmente nos vale con poder conectar el cable a la tarjeta (siempre que tenga asignación automática de IPs, DHCP activo), no hay que hacer nada más para poder comenzar a utilizar la conexión, sin embargo no es así en el caso de las conexiones WiFi, donde necesitamos su identificador de red, su SSID (*Service Set IDentifier*, Identificador de Conjunto de Servicios) o bien lo que normalmente conocemos como nombre de la red; también lo normal es que la red se encuentre protegida con algún tipo de contraseña, por lo que la función de inicialización de la conexión en caso de ser WiFi debe tener estos aspectos en cuenta. Véase la figura 12.5.
Para poder trabajar con la librería Wifi, se debe añadir al sketch mediante el menú Sketch>Importar librería...>WiFi o bien añadiendo la línea de código siguiente:

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

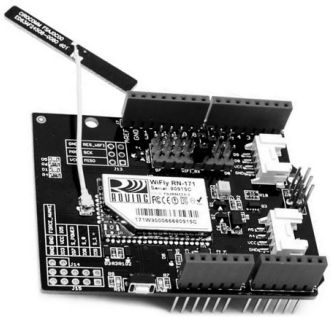


Figura 12.5. *Shield* por WiFi.

A grandes rasgos, la librería WiFi para nosotros como desarrolladores, funciona de modo semejante a la librería Ethernet, de la cual ya hemos visto el funcionamiento, aunque algún aspecto varía; por ejemplo, algo que no teníamos que hacer que hacer al utilizar la librería Ethernet era preocuparnos por enlazarnos con la red, pero en caso de WiFi sí. La iniciación de la conexión a la red la realizaríamos mediante la llamada:

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

Utilizando como parámetros el identificador de la red y la clave de ésta. Por ejemplo:

```
#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

En caso de trabajar con conexiones GSM pasa algo semejante al caso WiFi; tendremos una serie de funciones comunes con la librería Ethernet, pero otras serán distintas; por ejemplo, cuando encendemos un móvil necesitamos introducir (normalmente) un pin antes de comenzar a usarlo; aquí sucede lo

mismo, antes de poder trabajar con las funciones de la tarjeta es necesario validarse mediante su pin. Para trabajar con la librería GSM podemos introducir la línea de código:

```
#include <GSM.h>
```

o hacerla accesible mediante el menú Sketch>Importar librería...>GSM.

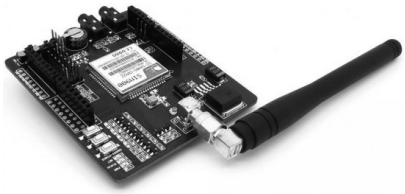


Figura 12.6. *Shield* por GSM.

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

```
gsm.begin(pin)
```

donde `pin` no es ningún terminal de Arduino, sino el número pin de identificación de la tarjeta. Por ejemplo:

```
#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!");
}
```

Además se debe tener en cuenta que se está trabajando con una tarjeta de teléfono, por lo que existen funciones para controlar el estado de éste; por ejemplo `gsm.getVoiceCallStatus()` para conocer el estado de la llamada de voz o `hangCall()` para colgar.
En caso de utilizar la *shield* GSM tenemos a nuestra disposición otras clases interesantes como `GSM_SMS` que facilita el envío de mensajes SMS.

13 Memorias

En este capítulo aprenderá a:

- Diferenciar las distintas memorias disponibles.
- Conocer diferentes métodos de almacenamiento.
- Gestionar la memoria de los programas.
- Utilizar las tarjetas SD como almacenamiento.
- Leer datos de una tarjeta SD.

Las placas Arduino poseen tres tipos de memoria disponibles para trabajar con ellas:

- Memoria Flash que es donde se almacena el *sketch* para poder ser ejecutado por el microcontrolador. Esta memoria no se borra al perder la alimentación de la tarjeta.
- SRAM (*Static Random Access Memory*, memoria estática de acceso aleatorio) que es donde se crean, guardan y manipulan las variables al ejecutarse el *sketch.* Todos los datos almacenados en este tipo de memoria se pierden cada vez que la tarjeta se desconecta de alimentación.
- EEPROM (*Electrically ErasableProgrammable Read-Only Memory*, Memoria de solo lectura borrable programable eléctricamente) que es donde los programadores pueden almacenar información que persistirá incluso sin alimentación; es decir, datos de larga duración.

La cantidad disponible de cada uno de los tipos de memoria descritas anteriormente, depende de los modelos de tarjeta y de su microcontrolador:

Tipo memoria	Arduino Uno (ATmega328)	Arduino Mega (ATmega2560)
Flash	32kb	256kb
SRAM	2kb	8kb
EEPROM	1kb	4kb

Dentro de la memoria Flash, hay un espacio que se reserva para el *bootloader* de Arduino, siendo 500 bytes para los chips ATmega328 y 8 kbytes para los ATmega2560, lo que significa que no podemos ocupar toda la memoria con el programa. Por otro lado, tenemos que para la ejecución del programa disponemos de 2048 bytes en el caso de Arduino Uno, cuando el programa se ejecute se comenzarán a crear las variables dentro de este espacio y comenzarán a ocuparlo. Las variables de tipo array por su carácter dimensional pueden ser las más preocupantes a la hora de su alojamiento en memoria; supongamos la sentencia:

```
char txt[ ] = "Esto de Arduino es muy interesante";
```

durante la ejecución ocuparía 35 bytes de la SRAM (no hay que olvidar el carácter fin de cadena *null* que se introduce de modo automático), por lo que si usamos excesivas cadenas de texto es muy probable que llenemos toda la memoria y el resultado sea totalmente imprevisible; nos daremos cuenta de que estamos sin memoria porque el programa no se ejecuta o no responde

como debería. En programas en los que la memoria pueda ser algo limitador, se debe tener especial cuidado en el tipo de variables que se usan, por ejemplo usando siempre que sea posible tipos `byte` en lugar de `int` (1 byte en memoria frente a 2 bytes que ocupa un `int`); también se pueden utilizar cadenas de texto más cortas o descargar parte de la lógica y datos a otros sistemas, por ejemplo si interactuamos con un móvil, se puede enviar desde Arduino un código de mensaje y que la descripción del mensaje se encuentre en el propio móvil en lugar de enviar el mensaje completo.
Si todo esto no fuera suficiente debemos de recurrir a medidas un poco más drásticas y usar otras memorias para acomodar todos los datos necesarios para el funcionamiento del programa. Veamos algunas soluciones.

Memoria Flash

Como se ha comentado anteriormente, la memoria Flash se utiliza para almacenar el *sketch* que se va a ejecutar, pero como podemos observar, la cantidad de memoria disponible de tipo Flash es muy superior a la disponible de tipo SRAM, por lo que podemos aprovechar que muchas veces las variables ya están definidas en el programa, para utilizarlas directamente desde allí, sin necesidad de transportarlas a la SRAM y por lo tanto ocupar espacio en esta memoria. Supongamos que tenemos una cadena de texto que mostraremos en el monitor, pues en lugar de moverla hacia la SRAM y que el *sketch* durante la ejecución tome el dato de esta memoria, haremos que la tome directamente de la memoria Flash donde ya se encuentra almacenada. Para ello usaremos el modificador de variables `PROGMEM`, al definir una variable con esta palabra, indicamos que no se lleve al espacio de ejecución SRAM, sino que se trabaje sobre ella desde Flash. Un ejemplo sería:

```
tipoDato nombreVariable[] PROGMEM = {};
```

aunque también se podría escribir:

```
PROGMEM tipoDato nombreVariable[] = {};
```

No todos los tipos de variables se pueden mantener en la memoria Flash, sólo los que están definidos en el archivo de cabecera `<avr/pgmspace.h>`, pero para tranquilidad del lector comentar que están los más habituales `char`, `int` de distintos tamaños, `void` ... lo malo es que no están definidos como los venimos usando en Arduino, sino que tienen sus propios tipos. Por ejemplo para almacenar un array entero con signo de dos bytes sería:

```
PROGMEM  prog_int16_t myData[]  = { -134, 2761, -1843, 10, 1,1234};
```

A la hora de recuperar también se debe hacer de un modo especial, mediante funciones definidas en el propio fichero de cabecera <avr/pgmspace.h>, por ejemplo para acceder al tercer elemento del array usaríamos:

```
int myInt = pgm_read_word_near(myData + 2);
```

Vamos a realizar un pequeño ejemplo donde veremos cómo utilizar este tipo de memoria, pero sin más utilidad que la puramente académica. Realizaremos un *sketch* en el que guardaremos algunos datos en la memoria Flash y luego los recuperaremos mostrando su salida en el monitor serie.
Para poder trabajar con la memoria Flash lo primero es importar el fichero de cabecera.

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

Después definiremos los datos que se utilizarán directamente desde la memoria Flash así como un buffer que nos ayudará en la lectura de las cadenas de texto.

```
PROGMEM prog_int16_t signedInts[]  = {134, 2761, -1843, 10, 1,1234};
PROGMEM prog_char text[]   = "Esto de Arduino es muy divertido.";
char buffer[50];
```

Y por último debemos configurar el programa para que se comunique con el monitor serie y en el bucle principal ir llamando a las distintas funciones para leer los datos almacenados. El *sketch* sería:

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

PROGMEM prog_int16_t signedInts[]  = {134, 2761, -1843, 10, 1,1234};
PROGMEM prog_char text[]   = "Esto de Arduino es muy divertido.";

char buffer[50];
void setup(){
   Serial.begin (9600);
}

void loop(){
   // obtención del signedInts que son enteros con signo
   Serial.print("Tercer valor de signedInts: ");
   int value = pgm_read_word (signedInts + 2);
   Serial.println(value, DEC);
   delay(500);

   // obtención de error por trabajar sin signo
   Serial.print("Tercer valor ERRONEO de signedInts: ");
   Serial.println(pgm_read_word (signedInts + 2), DEC);
   delay(500);

   // lectura de parte del texto
   Serial.print("Segunda letra  de text: ");
   char secondChar =  pgm_read_byte (text + 1);
```

```
    Serial.println(secondChar);
    delay(500);

    // lectura de todo el texto
    strcpy_P(buffer,text);
    Serial.print("Valor del texto: ");
    Serial.println( buffer );
    delay(1000);
}
```

Hay que tener en cuenta los tipos de datos con los que se trabaja, por ejemplo en el segundo caso de lectura, estamos leyendo un dato que es con signo y lo estamos tratando como si fuera sin signo, lo que dará una representación errónea. Mediante `pgm_read_word(dirección)` o `pgm_read_word_near(dirección)`, realizamos la lectura de una palabra (`word`, 2 bytes) a partir de la dirección indicada. Del mismo modo utilizamos `pgm_read_byte(dirección)` o `pgm_read_byte_near(dirección)` para la lectura de un sólo byte, como es el caso del carácter. Para la lectura de toda una cadena, la situación se complica un poco más; lo que hacemos es copiar el valor de la variable en Flash a SRAM (es decir, no pasa a SRAM hasta que no es necesario) mediante la función `strcpy_P(destino, origen)` donde `origen` es la dirección de memoria a copiar y `destino` la dirección donde copiar (que está en SRAM). Copiará hasta encontrar la marca de fin de cadena.

Advertencia:

Los datos almacenados en la memoria Flash no pueden ser modificados durante la ejecución del sketch.

En cuanto a las cadenas de texto que se usan por ejemplo en las salidas del monitor del ejemplo anterior:

```
Serial.print("Segunda letra de text: ");
```

el funcionamiento es idéntico, se llevan al espacio de ejecución SRAM y desde ahí se envían mediante la conexión serie, es decir, también ocupan memoria. Podemos crear una constante para cada una de ellas y utilizar el método visto anteriormente para hacer que se almacenen en la Flash, pero sería muy tedioso y largo de realizar. Para facilitarnos la tarea, existe una función que nos permitirá indicar que se guarde el dato en la memoria Flash y sin variar mucho el código, se trata de `F()` que está disponible desde la versión 1.0 de Arduino; así por ejemplo para mantener en Flash el texto de la línea anterior de código se escribiría:

```
Serial.println(F("Segunda letra de text: "));
```

Se habrá fijado el lector que hemos visto lectura pero no escritura, y es que los datos mantenidos en Flash son datos que no se modificarán a lo largo de la ejecución del programa; en caso de querer modificarlos, se debe llevar el dato a una variable alojada en la SRAM y modificarlo.

Memoria EEPROM

Hemos visto que es posible que en ocasiones necesitemos almacenar datos en la memoria Flash por falta de espacio en la memoria SRAM, pero tenemos el problema que son datos que solamente pueden ser leídos, no pueden modificarse de modo alguno. Otro tipo de memoria que tiene la placa Arduino es la EEPROM (*Electrically ErasableProgrammable Read-Only Memory*, Memoria de sólo lectura borrable programable eléctricamente), que aunque de pequeño tamaño (dependiendo del modelo varía: 1024 bytes en los ATmega328, 512 bytes en ATmega168 y ATmega8, 4096 bytes en ATmega1280 y ATmega2560) puede tener gran utilidad ya que es posible leer y escribir en ella (como en la SRAM), pero además sus datos se mantienen aunque se elimine la alimentación de la placa, es decir es como una especie de pequeño disco duro donde podremos guardar nuestros datos y estos permanecerán allí entre distintas ejecuciones del *sketch*.

Para trabajar con la memoria EEPROM disponemos de una librería estándar que da funcionalidad limitada y una librería extendida que no es parte del estándar Arduino, pero para aquellos que quieran hacer uso intensivo de la EEPROM es muy recomendable. La librería extendida se denomina EEPROMex (disponible en `http://thijs.elenbaas.net/downloads/`) y es una variación de la librería estándar de Arduino que permite la lectura y escritura de tipos básicos, mientras que la estándar simplemente permite lectura y escritura de bytes. Nosotros nos centraremos en la librería estándar de Arduino; trabajaremos byte a byte.

Para ver su funcionamiento haremos un pequeño ejemplo que muestre en pantalla un texto en mayúsculas o en minúsculas dependiendo de la pulsación de un botón y utilizaremos la memoria EEPROM para guardar el estado entre diferentes ejecuciones del *sketch*, como si fueran unas preferencias de usuario. El cambio entre mayúsculas y minúsculas vendrá dado por la pulsación de un botón. Para el montaje electrónico necesitaremos simplemente un pulsador. Véase la figura 13.1.

En el *sketch* lo primero que se debe realizar como en ocasiones anteriores es incluir el archivo de cabecera correspondiente para facilitar los desarrollos, en este caso es el `EEPROM.h`.

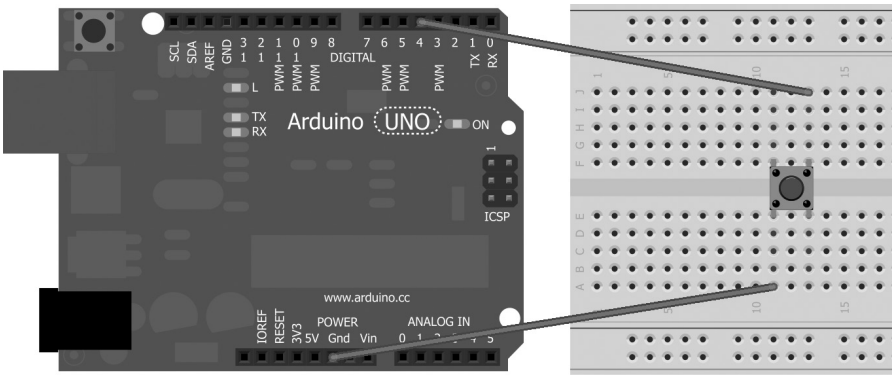


Figura 13.1. Montaje para pruebas con EEPROM.

Utilizaremos un pin de entrada para poder leer el estado del botón y detectar así si se quiere que se haga el cambio de mayúsculas a minúsculas y viceversa. Para evitar que al mantener apretado el botón se pase de mayúsculas a minúsculas en cada ciclo, usaremos una técnica ya conocida y usada en ejercicios anteriores para detectar que se ha dejado de pulsar el botón antes de realizar otra lectura sobre éste. En caso de que se detecte pulsación se realizará el cambio del valor de la variable guardada en la EEPROM mediante la función `changeCaps()`; esta función se encargará de leer el valor actual guardado en la EEPROM, que será un *booleano*, cambiarlo de valor y volverlo a guardar en la en esta memoria.

```
void changeCaps(){
   boolean value = EEPROM.read(0); // lee el valor de la EEPROM
   EEPROM.write(0,!value); // salva el valor en la EEPROM
}
```

La lectura del valor de la EEPROM se hace con `EEPROM.read(dirección)` y en este caso tomamos como dirección a leer la 0 (leemos el primer byte), una vez leída se guarda el contrario de lo que se ha leído mediante `EEPROM.write(dirección,valor)` donde `dirección` es la dirección de memoria donde escribir y `valor` es el valor del byte a escribir (se debe tener cuidado de no pisar los datos cuando se graban nuevas entradas en la EEPROM vigilando los bytes que ocupa cada dato).
El *sketch* terminado sería:

```
#include <EEPROM.h>

const int ledPin = 13; // pin del led
const int inputPin = 4; // pin para el botón
boolean caps = false; // marca si se deben usar las mayúsculas
boolean checkButton = true;
```

```
void setup(){
   Serial.begin (9600);
   pinMode(ledPin, OUTPUT); // pin del led como salida
   pinMode(inputPin, INPUT); // se declara como pin de entrada
   digitalWrite(inputPin, HIGH); // se activa la resistencia de pull-up
}

void loop(){
   // lee la configuracion
   int value = EEPROM.read(0);

   // si es verdadero, escribe en mayúsculas
   if (value){
      Serial.println("ESCRIBE EN MAYUSCULAS");
   }
   else{
      Serial.println("escribe en minuscula");
   }
   // led encendido si mayúsculas
   digitalWrite(13,value);

   if (!digitalRead(inputPin)){ // ver si esta pulsado
      if (checkButton){ // se debe atender la pulsación?
         checkButton = false;    // dejar de atender pulsaciones si se
                                 // atiende una
         changeCaps();
      }
   }
   else{
      checkButton = true; // se ha soltado el botón, volver a atenderlo
   }

}

/** Cambia el valor del byte 0 de la EEPROM negándolo **/
void changeCaps(){
   boolean value = EEPROM.read(0); // lee el valor de la EEPROM
   EEPROM.write(0,!value); // salva el valor en la EEPROM
}
```

En el cuerpo principal del *sketch* realizamos una lectura del valor almacenado en la EEPROM para conocer si el mensaje se debe mostrar en mayúsculas o minúsculas y la lectura se produce en cada ocasión, este método no es el más ortodoxo para una programa real pero si para este caso didáctico; en el caso real podríamos leerlo solamente en el `setup()` y luego mantenerlo en una variable, y eso sí, guardarlo del mismo modo que en este caso.

Si se trabajara con datos que ocupen más de un byte, se debe tener cuidado a la hora de salvar y recuperar, que se haga en el mismo orden y lo mismo pasa con las cadenas de texto, que son una sucesión de bytes. No obstante si uno no se siente cómodo trabajando byte a byte, hay que recordar que se puede echar mano a las librerías extendidas.

Memoria externa

En ocasiones puede que toda la memoria disponible en la tarjeta Arduino no sea suficiente para almacenar todos los datos necesarios por nuestra aplicación o bien puede que queramos mantener un registro a modo de diario de las acciones realizadas por el circuito de automatismos que tengamos conectado al microcontrolador; para ello podemos valernos de memorias externas y almacenar sobre ellas todo el contenido que necesitemos.

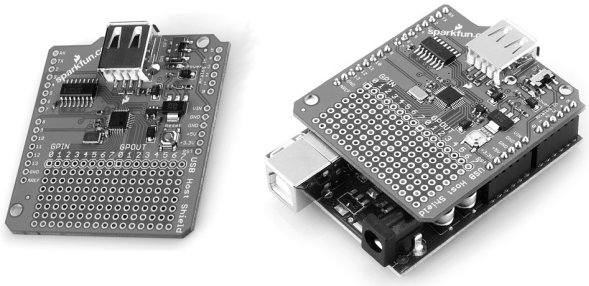


Figura 13.2. *Shield* USB Host.

Para poder almacenar datos en memorias externas podemos optar por las soluciones más extremas como puede ser crear un interfaz para discos duros IDE (actualmente hay varios proyectos en este sentido) o ir más por lo práctico y decantarse nuevamente por alguna de las *shields* disponibles en para esta función.

Dentro de las *shields* que hay en el mercado para ofrecer almacenamientos externos las más comunes que podemos encontrar son las de tipo USB Host y las que son compatibles con tarjetas SD. Las de tipo USB Host permiten usar USBs de memoria, discos duros e incluso algunas cuentan con la implementación de las especificaciones dictados por ADK (*Android Development Kit*, Kit de desarrollo Android) que permiten comunicar dispositivos Android con otro tipo de hardware vía USB. En cuanto a las compatibles con tarjetas SD, las hay también de múltiples tipos: para tarjetas SD, miniSD, microSD, combinaciones de ellas... A la hora de decantarse por la compra de alguna *shield*, hay que mirar las opciones disponibles y que más nos convengan, puesto que muchos fabricantes suelen ofrecer varias soluciones dentro de la misma *shield*; por ejemplo podemos encontrar lector de SD con USB host y Ethernet en una misma tarjeta.

Pasaremos a ver un poco más en profundidad la opción del manejo de datos en SD.

Memorias SD

Las tarjetas SD (*Secure Digital*) son muy conocidas por encontrarse en las cámaras fotográficas digitales y en los primeros teléfonos móviles de pantalla táctil (allá por el 2004). A lo largo del tiempo, su tamaño se ha ido reduciendo a la vez que su capacidad iba aumentando y muchos eran los fabricantes que ofrecían este producto; para que todos estos fabricantes siguieran unas especificaciones concretas, en el año 2000, una serie de fabricantes (entre los que estaban Panasonic, SanDisk y Toshiba entre otros) fundan la SD Association, que se encargará de dictar las especificaciones y protocolos a utilizar en este tipo de almacenamiento.
Las medidas de las tarjetas SD que encontramos hoy en día están completamente regladas siendo de 32x24mm para la SD original, 21x20mm para la miniSD y 15x11mm para la microSD. En cuanto a las capacidades éstas varían desde unos megas de las primeras SD hasta los 2 terabytes de las SDXC.

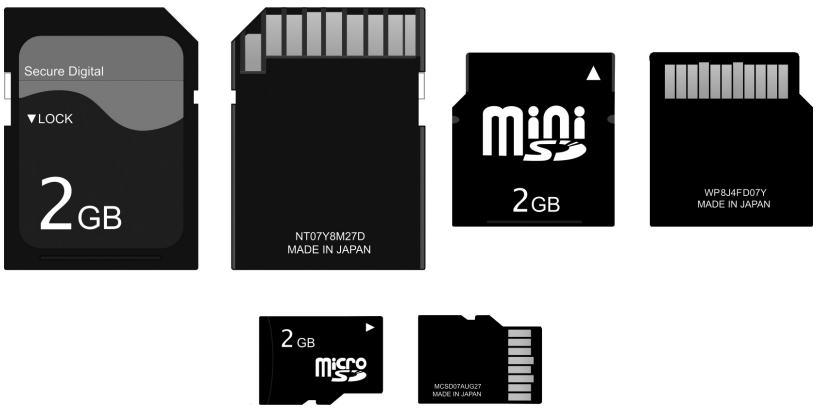


Figura 13.3. Tarjetas SD.

Cuando se utilizan estas tarjetas, se debe vigilar que sean compatibles con el lector, dado que aunque existe la SD Association para dictar las especificaciones, en ocasiones los fabricantes son más rápidos y generan sus propias especificaciones o protocolos que luego no son compatibles con los estándares, por ejemplo como ocurrió cuando empezaron a salir las primeras tarjetas de más de 1024 megas, que no existía aún una estandarización sobre el acceso a datos y cada fabricante implementaba el suyo (actualmente ya está reglado). En cuanto a compatibilidad con versiones anteriores de las tarjetas, no debemos preocuparnos por los lectores, ya que suelen fabricarse que sean compatibles con las tarjetas más antiguas; pero si el caso es el contrario, que

disponemos de un lector antiguo y una tarjeta nueva, entonces es posible que tengamos algunos problemas ya que puede que no lea toda la capacidad de la tarjeta o no interprete bien su contenido.
Para poder utilizar las tarjetas de menor tamaño físico con los lectores preparados para las de mayor tamaño (por ejemplo usar una tarjeta microSD con lector de SD), existen unos adaptadores pasivos donde se introduce la tarjeta menor y este adaptador a modo de funda hace que pueda acoplarse al lector en cuestión.
De aquí en adelante nos referiremos a todas estar tarjetas como tarjetas SD o simplemente SD abarcando tanto las SD estándar como las miniSD y las microSD, el lector será quien decida con qué tipo de tarjeta quiere trabajar dependiendo de disponibilidad de tarjeta y lector, ya que la programación y el montaje electrónico no depende del modelo seleccionado.

Información de la tarjeta SD

Para poder trabajar con las tarjetas SD desde Arduino, disponemos de dos opciones: comprar el lector de tarjetas y conectarlo nosotros mismos u optar por un montaje ya preparado; utilizar una *shield*. Las *shield* de lectura de tarjetas suelen traer además otras utilidades incorporadas (como lectores de varios tamaños de SD o Ethernet) y a precios muy asequibles por lo que normalmente sale a cuenta.

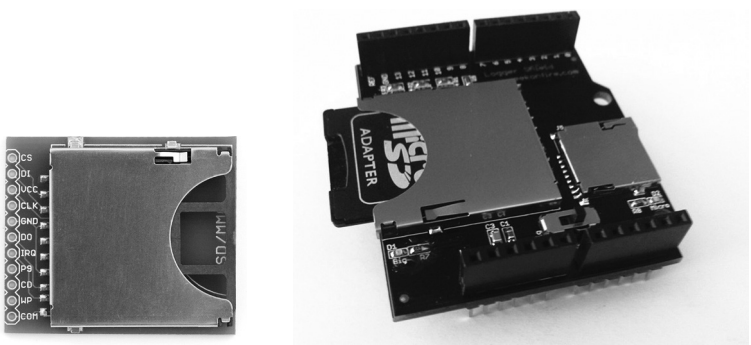


Figura 13.4. Lector de tarjetas SD y *shield*.

Vamos a ver a continuación una serie de ejemplos para ilustrar el uso de las tarjetas SD. Dependiendo de si finalmente se ha optado por el lector suelto o por una *shield*, el montaje electrónico será diferente. En caso de la *shield* debemos preocuparnos solamente de que todas las patillas de los terminales

encajen correctamente en la placa Arduino. Si se usa un lector suelto, deberemos hacer un poco más de trabajo, lo primero es que necesitaremos más materiales a parte del lector, que serán tres resistencias de 3.3kΩ y tres resistencias de 1.8kΩ; también se debe poner especial cuidado a la hora de realizar las conexiones y alimentar al lector, ya que normalmente son alimentados a 3.3V pero existe alguno de 5V. Las conexiones con los terminales de la tarjeta Arduino son las siguientes:

Pin Lector	Pin Arduino	Pin Arduino Mega
GND	GND	
VCC	5V o 3.3V dependiendo del lector	5V o 3.3V dependiendo del lector
CS	10	53
DI o MOSI	11	51
DO o MISO	12	50
CLK o SCK	13	52

El pin de selección de tarjeta (pin CS) puede variar dependiendo de la *shield* que utilicemos, por ejemplo en la *shield* Ethernet que incorpora lector de tarjetas SD, dado que se utiliza el pin 10 para otros menesteres, se ha movido la funcionalidad al pin 4 o la *shield* de la empresa Sparkfun lo tiene localizado en el pin 8, por lo que si se utilizan algunas de estas *shield* se deben tener en cuenta a la hora de realizar los ejemplos y configurarlo de modo correcto, de lo contrario no podremos inicializar la tarjeta.
En cuanto al sistema de archivos utilizado, lo más común es usar el archiconocido sistema FAT de Microsoft, así pues si nuestra tarjeta se encuentra en otro formato de archivos, se debe formatear a FAT para poder trabajar con ella. Como es posible que ya hayamos olvidado lo que quiere decir trabajar con sistema FAT, cabe recordar que los nombres de archivos siguen la convención de tener 8 caracteres para el nombre en sí del fichero, un punto y otros tres caracteres para la extensión que suele indicar el tipo de archivo que es (que no tiene porque serlo, técnicamente podemos guardar un jpg con extensión mp3), por ejemplo "mitexto.txt". Véase la figura 13.5.
Para los ejemplos, si se va a utilizar un lector suelto en lugar de una *shield*, debemos asegurarnos a qué voltaje se trabaja. En la figura del circuito hemos supuesto que trabajaríamos con un lector a 3.3V, por eso existen unos divisores de tensión en cada uno de los terminales de entrada de datos; Arduino

nos brinda las salidas a 5V y necesitamos que la entrada sea a 3.3V así pues realizamos un divisor con las resistencias de de 3.3kΩ y de 1.8kΩ; la salida obtenida así es de $V_{sal} = V_{ard} * \frac{R_1}{R_1 + R_2} = 5V * \frac{3300\Omega}{3300\Omega + 1800\Omega} = 3.24V$, que es un poco menos que los 3.3V, que se conseguirían con una resistencia de 1.7kΩ pero no es una resistencia estándar, así que las resistencias de 1.8kΩ ya nos valen.

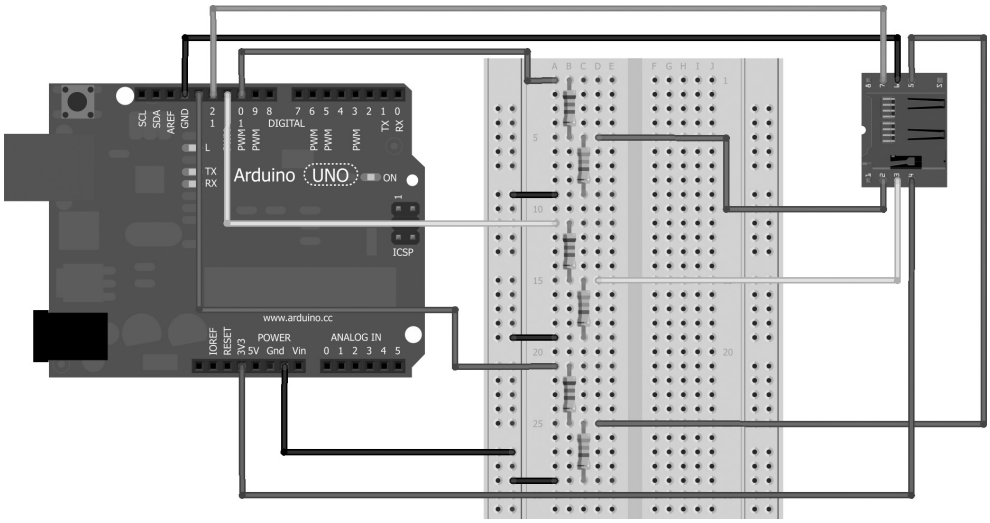


Figura 13.5. Circuito con lector de tarjetas SD.

Antes de ponernos nosotros a hacer un *sketch*, comenzaremos viendo uno que viene de ejemplo con Arduino que es muy sencillo y a la vez completo y muestra información de la tarjeta y un listado de los ficheros que contiene. Para cargarlo debemos seleccionar el menú Archivo>Ejemplos>SD>CardInfo, y obtendremos en pantalla un *sketch* semejante al siguiente listado, que ha sido ligeramente modificado.

```
#include <SD.h>

// variables a utilizar durante la adquisición de información
Sd2Card card;
SdVolume volume;
SdFile root;

// pin CS, cambiar dependiendo de la tarjeta
// Arduino Ethernet shield: pin 4
// Adafruit SD shields y modulos: pin 10
// Sparkfun SD shield: pin 8
const int chipSelect = 10;

void setup(){
```

```
Serial.print("\nIniciando tarjeta SD...");
// pin SS
pinMode(10, OUTPUT);     // 53 en modelo Mega

// Confirmar que se inicia bien la tarjeta
if (!card.init(SPI_HALF_SPEED, chipSelect)) {
   Serial.println("La iniciación de la tarjeta ha fallado. ");
   Serial.println("Comprobar que todo está correctamente conectado y el _
         pin CS seleccionado de modo correcto ");
   return;

} else {
   Serial.println("Todo ha ido bien.");
}

// print the type of card
Serial.print("\nTipo de tarjeta: ");
switch(card.type()) {
   case SD_CARD_TYPE_SD1:
      Serial.println("SD1");
      break;
   case SD_CARD_TYPE_SD2:
      Serial.println("SD2");
      break;
   case SD_CARD_TYPE_SDHC:
      Serial.println("SDHC");
      break;
   default:
      Serial.println("Desconocida");
}

// abrimos el volumen
if (!volume.init(card)) {
   Serial.println("No se ha podido encontrar un volumen FAT16/32 _
         correcto.");
   return;
}

// se imprime la información del volumen
uint32_t volumesize;
Serial.print("\nTipo de volumen FAT");
Serial.println(volume.fatType(), DEC);
Serial.println();

volumesize = volume.blocksPerCluster();
volumesize *= volume.clusterCount();
volumesize *= 512;
Serial.print("Volume size (bytes): ");
Serial.println(volumesize);
Serial.print("Volume size (Kbytes): ");
volumesize /= 1024;
Serial.println(volumesize);
Serial.print("Volume size (Mbytes): ");
volumesize /= 1024;
Serial.println(volumesize);
```

```
   Serial.println("\nFicheros en la tarjeta (nombre, fecha y longitud _
         en bytes): ");

   root.openRoot(volume);

   // listar contenido y longitude de fichero
   root.ls(LS_R | LS_DATE | LS_SIZE);
}

void loop(void) {

}
```

El lector puede utilizar directamente el obtenido en el entorno de desarrollo o utilizar el código anterior.
Lo primero que nos debe extrañar es que la función `loop()` se encuentra vacía y que todo el código está en la función `setup()`; esto es porque lo que se quiere ejecutar sólo se debe hacer una vez, si estuviera en el `loop()`, se ejecutaría indefinidamente; no hay que pasar por alto que es un ejemplo didáctico.
La librería `SD.h` utiliza el protocolo SPI (*Serial Peripheral Interface*, interfaz periféricos serie) para comunicarse con la tarjeta; se trata de un protocolo síncrono de transmisión de datos usado por algunos microcontroladores para la comunicación a corta distancia con periféricos u otros microcontroladores. Los pines involucrados en esta transmisión son:

- Pin MOSI: (*Master Out Slave In*, Maestro Sale Esclavo Entra) es el pin utilizado para el envío de datos del maestro al esclavo; del microcontrolador a los periféricos.
- Pin MISO: (*Master In Slave Out*, Maestro Entra Esclavo Sale) es el pin utilizado para el envío de datos del esclavo al maestro; del periférico al microcontrolador.
- Pin CLK: (*Clock*, Reloj) es el reloj que sincronizará mediante sus pulsos las transmisiones del maestro. Este pin también puede aparecer como SCK (*Serial Clock*, Reloj serie).
- Pin SS: (*Slave Select*, Selector de Esclavo) el pin utilizado para habilitar y deshabilitar cada periférico. Es necesario uno por cada periférico que se quiera conectar como esclavo. Cuando se encuentra a `LOW` el esclavo puede comunicarse con el maestro; si está a `HIGH` el esclavo es ignorado. Usando la activación y desactivación de los esclavos podemos hacer uso de los pines *MOSI*, *MISO* y *CLK* de modo común para todos ellos, y simplemente habilitaremos aquél esclavo con el que queremos comunicar. Pero volvamos al código.

La librería que nos da acceso a toda la funcionalidad sobre las tarjetas SD la importamos mediante:

```
#include <SD.h>
```

A partir de este momento tendremos accesibles múltiples funciones para trabajar con tarjetas formateadas en FAT32 y FAT16; eso sí con ficheros con nombres 8.3, no lo olvidemos. En cuanto a los tipos de tarjeta soportados se encuentran las tarjetas SD y las tarjetas SDHC.
Justo antes de la función `setup()`, tenemos la configuración del pin que se utilizará como selector del chip y que nos servirá para inicializar la tarjeta. Dependiendo del modelo a usar se debe modificar este valor. En los comentarios del programa explica algunos de los posibles valores para las placas más utilizadas.

```
const int chipSelect = 10;
```

Dentro de la función `setup()`, configura el monitor serie y el pin SS que siempre se debe dejar en modo salida para trabajar con las SD, de lo contrario la librería no funcionará de modo correcto. Recordemos que dependiendo de las tarjetas el pin SS cambia.

```
pinMode(10, OUTPUT);
```

Lo siguiente que se realiza es la inicialización de la tarjeta para poder trabajar con ella.

```
if (!card.init(SPI_HALF_SPEED, chipSelect)) {
```

En este caso se inicializa con la variable `card` que es de clase `Sd2Card`, indicando en su método de inicialización, tanto la velocidad a la que se realizará la comunicación como el pin encargado de seleccionar el hardware esclavo. Las velocidades pueden ser `SPI_FULL_SPEED`, `SPI_HALF_SPEED` o `SPI_QUARTER_SPEED` y es un parámetro optativo al igual que `chipsSelect`, que podemos informarlo o no (aunque debe estar en modo `OUTPUT`).
Dependiendo de si la tarjeta no ha sido convenientemente inicializada se muestra un mensaje de error y se sale o si se ha inicializado bien se muestra mensaje confirmándolo y se continúa con el programa leyendo el tipo de tarjeta que es; esto se realiza mediante:

```
switch(card.type()) {
```

Esta llamada nos puede devolver los siguientes tipos *SD V1, SD V2* o *SDHC* que son comparados con sus correspondientes constantes mediante el `swith()` y se muestra en el monitor el tipo detectado.
Usando la variable `volume` de la clase `SdVolume` se intenta abrir el volumen de archivos de la tarjeta para más adelante obtener su información.

Al abrir el volumen se debe informar la tarjeta de la cual se quiere obtener el volumen de ficheros.

```
if (!volume.init(card)) {
```

Para recuperar el tamaño exacto del volumen de archivos, necesita realizar un par de operaciones matemáticas:

```
volumesize = volume.blocksPerCluster();
volumesize *= volume.clusterCount();
volumesize *= 512;
```

Mediante distintas llamadas obtenemos los bloques por grupo (bloques por *cluster*) y el número de grupos (*clusters*) disponibles en la tarjeta y así tener el tamaño total.
Por último se muestra el contenido de la tarjeta a través de la variable `root` de clase `SdFile`, que se inicializa pasándole como parámetro el volumen a leer, una vez inicializada se puede obtener su contenido mediante la función `ls()` que envía un listado de los ficheros a los pines de comunicación serie en nuestro caso el listado se muestra en el monitor serie.

```
root.openRoot(volume);
root.ls(LS_R | LS_DATE | LS_SIZE);
```

La función `ls()` tiene como parámetros unas constantes que modifican el contenido enviado al puerto serie: `LS_R` indica que se haga un listado recursivo de los directorios, `LS_DATE` sirve para mostrar la fecha de la última modificación del fichero y `LS_SIZE` nos muestra el tamaño del fichero.
Si ejecutamos el *sketch* obtendremos algo semejante a:

```
Inciando tarjeta SD ...Todo ha ido bien.

Tipo de tarjeta: SD2

Tipo de volumen FAT16

Volume size (bytes): 1015267328
Volume size (Kbytes): 991472
Volume size (Mbytes): 968

Ficheros en la tarjeta (nombre, fecha y longitud en bytes):
WIILOAD/      2009-12-06 19:50:56
   WIILOA~1.GZ   2009-10-07 22:05:30 59494
   WIN32/        2009-12-06 19:50:56
      WIILOAD.EXE   2009-10-07 22:05:30 61440
   OSX/          2009-12-06 19:50:56
      WIILOAD       2009-10-07 22:05:30 42968
   LIN32/        2009-12-06 19:50:56
      WIILOAD       2009-10-07 22:05:30 23710
BOOT.ELF      2009-10-07 22:05:30 1512992
CREDITS       2009-10-07 22:05:30 12867
LICENSE       2009-10-07 22:05:30 2199
```

```
README~1.TXT  2009-10-07 22:05:30 5313
README~2.TXT  2009-10-07 22:05:30 7425
NUEVOD~1.TXT  2009-12-12 09:03:04 0
PRIVATE/      2009-12-12 09:03:08
...
```

En caso de tener ficheros que se hayan guardado con un sistema operativo en el que estén permitidos nombres de ficheros mayores en tamaño que los 8.3 caracteres permitidos en estas tarjetas con tipo de archivos FAT, se mostrarán sus nombres cortados en el sexto carácter y completados con una vírgula y un número, por ejemplo `README~2.TXT`, que muestra un 2 porque ya existe un `README~1.TXT` y así los diferencia.

Escritura y lectura en tarjetas SD

Ahora que ya sabemos cómo obtener datos de nuestra tarjeta SD, vamos a almacenar en ella información que podremos recuperar más adelante.
En capítulo 8 vimos como realizar un circuito que medía la temperatura y humedad y se mostraban los datos en el monitor serie, en el capítulo anterior vimos como mostrarlo por Internet y en este capítulo aprovecharemos ese mismo circuito para modificarlo y el lugar de mostrar los datos, almacenarlos en un fichero de modo que más adelante pudiera leerse saberse qué temperatura y humedad ha hecho en cada momento, a modo de histórico.
Para el montaje electrónico necesitaremos un DHT21, una resistencia de 10kΩ y el lector de tarjetas SD.

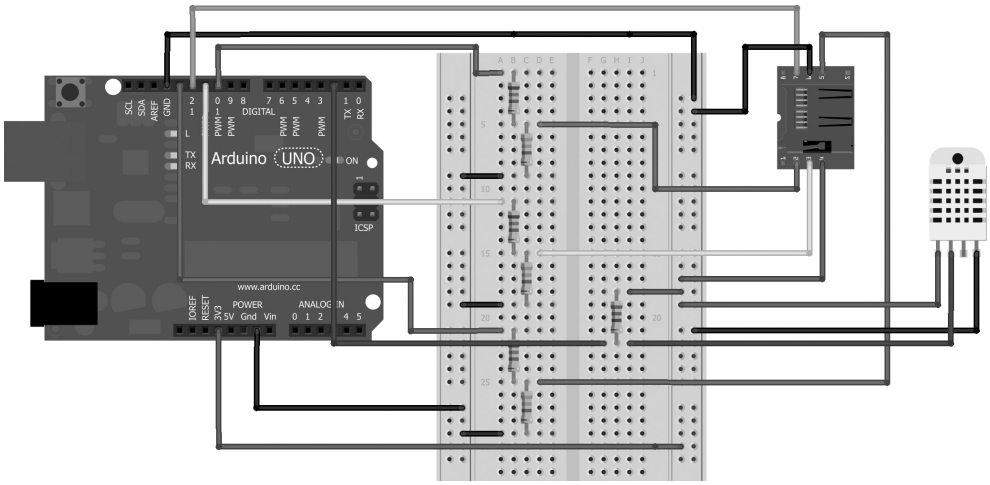


Figura 13.6. Circuito con lector de tarjetas SD y sensor DHT21.

El *sketch* con el que trabajaremos es también muy semejante al del ejemplo del capítulo 8, sólo que en este caso en lugar de mostrar la salida por el monitor serie, lo guardaremos también en la tarjeta SD.
Lo primero es incluir la librería SD mediante la línea:

```
#include <SD.h>
```

Necesitaremos una constante que guarde el pin para la selección de la tarjeta SD sobre la que escribir, que dependerá de cada modelo de tarjeta Arduino y de la *shield* utilizada; normalmente es el 10 pero se debe variar teniendo en cuenta las consideraciones explicadas anteriormente.

```
const byte cs= 10; // OJO ajustarlo a cada tarjeta
boolean hasSD; // guarda si tiene o no tiene tarjeta
```

Durante la función `setup()` se procede a la configuración e inicialización de la tarjeta SD, aprovecharemos para guardar si la inicialización ha sido correcta o no, para más adelante en la función `loop()` poder obviar la escritura en la tarjeta si hubo algún problema durante la inicialización. Para poder utilizar la tarjeta se llama a la función `begin()` de la clase `SD`, a la cual se le pasa como parámetro opcional el pin para la selección de la tarjeta a utilizar.

```
Serial.print("Iniciando tarjeta");
pinMode(cs, OUTPUT);
if (!SD.begin(cs)) {
   Serial.println("Tarjeta no presente");
   hasSD = false;
}
else{
   hasSD = true;
}
```

Por último, en la función `loop()` salvaremos los datos obtenidos en el fichero, dentro del case 0 del `switch()`, justo después de la salida del monitor serie:

```
// guardar en SD
if (hasSD){
   File dataFile = SD.open("templog.txt", FILE_WRITE);
   // si el fichero esta ok escribir en el
   if (dataFile) {
      char str[80];
      sprintf(str, "Humedad:%d.%d , Temperatura: %d.%d", _
            dhtData[0],dhtData[1],dhtData[2],dhtData[3]);
      dataFile.println(str);
      dataFile.close();
   }
   // si hay error mostrarlo en pantalla
   else {
      Serial.println("Error al abrir templog.txt");
   }
}
```

Guardamos los datos solamente si se inicializó correctamente la tarjeta (si no daría errores), esto lo controlamos mediante la variable `hasSD`. Si todo fue correcto, intentamos abrir el fichero `templog.txt` en modo escritura mediante la función `SD.open(nombre_fichero, modo_apertura)`, donde `nombre_fichero` es el nombre de fichero sobre el que se quiere trabajar y el `modo_apertura` indica si se quiere abrir en modo lectura (constante `FILE_READ`) o escritura (constante `FILE_WRITE`); este segundo parámetro es opcional y en el caso de no informarse por defecto se toma como que se abre en modo lectura.

Cuando se trabaja con la librería SD, los nombres de archivos se toman desde la raíz del árbol de directorios de la propia tarjeta, de modo que si se quiere escribir en un fichero `templog.txt` dentro del directorio `tmp`, nos debemos referir a él como `/tmp/templog.txt` o `tmp/templog.txt`, la primera barra "/" indicando el raíz del árbol es optativa, ya que siempre se toma desde el raíz. Aunque se trate de un sistema de archivos FAT, para separar directorios se deben utilizar las barras directas "/" y no las contra barras "\".

Una vez que el fichero está disponible para su uso, podemos escribir en él mediante métodos semejantes a los utilizados en el monitor serie, por ejemplo `println()` para escribir textos y añadir una nueva línea, `print()` para realizar la misma tarea pero sin nueva línea o `write()` para datos crudos.

Por último no debemos nunca olvidar cerrar los ficheros que hayamos abierto para trabajar con ellos, tanto en el caso de lectura como en escritura; el cierre del fichero se realiza mediante el método `close()`. Aunque utilicemos alguno de los métodos de escritura en la tarjeta, realmente la escritura sobre ella se realiza al llamar a las funciones `close()` o `flush()`, por lo que no debemos olvidar llamarlas o perderemos lo escrito. El *sketch* modificado para guardar los datos en la tarjeta SD quedaría:

```
#include <SD.h>
const byte dhtPin = 2;
byte errorCode; // código de error
byte dhtData[5]; // bytes leídos
const byte cs= 10; // OJO ajustarlo a cada tarjeta
boolean hasSD; // guarda si tiene o no tiene tarjeta

void setup(){
   pinMode(dhtPin,OUTPUT);
   digitalWrite(dhtPin,HIGH);
   Serial.begin(9600);
   Serial.print("Iniciando tarjeta");
   pinMode(cs, OUTPUT);
   if (!SD.begin(cs)) {
      Serial.println("Tarjeta no presente");
      hasSD = false;
   }
```

```
   else{
      hasSD = true;
   }
}

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.");
         // guardar en SD
         if (hasSD){
            File dataFile = SD.open("templog.txt", FILE_WRITE);
            // si el fichero esta ok escribir en el
            if (dataFile) {
               char str[80];
               sprintf(str, "Humedad:%d.%d , Temperatura: %d.%d", _
                     dhtData[0],dhtData[1],dhtData[2],dhtData[3]);
               dataFile.println(str);
               dataFile.close();
            }
            // si hay error mostrarlo en pantalla
            else {
               Serial.println("Error al abrir templog.txt");
            }
         }
         break;
      // Errores
      case 1:
         Serial.println("Error 1: Ha fallado la primera condicion 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;
}
```

Al ejecutar este *sketch*, se comenzarán a guardar tanto la temperatura como la humedad en el fichero `templog.txt` situado en la raíz de la tarjeta SD.

Advertencia:

Al abrir un fichero tanto para lectura como para escritura no nos debemos olvidar de cerrarlo mediante la llamada al método close() *de la clase* File.

Para ver el resultado podemos utilizar el lector de tarjetas de nuestro ordenador o móvil o realizar un *sketch* que se encargue de mostrar el resultado en el monitor serie. Como ya tenemos soltura suficiente, vamos a optar esta última opción. Sin modificar el circuito electrónico (o si se quiere se puede desconectar el sensor de humedad) haremos un *sketch* que lea el contenido creado en el ejemplo anterior. El programa será muy semejante a la parte añadida en el código recientemente visto; deberemos abrir el fichero en modo lectura y proceder a leerlo a la vez que se iría volcando la información leída en el monitor serie. Dado que sólo nos interesa leer una vez el archivo, colocaremos el código en la función `setup()` en lugar de en la función `loop()`.

```
#include <SD.h>

const byte cs = 10; // OJO ajustarlo a cada tarjeta

void setup(){
```

```
   Serial.begin(9600);

   Serial.println("Iniciando tarjeta...");
   pinMode(cs, OUTPUT);

   if (!SD.begin(cs)) {
      Serial.println("Tarjeta no presente.");
      return;
   }

   Serial.println("Tarjeta inicializada.");
   // abrimos el fichero en modo lectura
   File dataFile = SD.open("tempwlog.txt",FILE_READ);

   // si existe leer los datos
   if (dataFile) {
      while (dataFile.available()) {
         Serial.write(dataFile.read());
      }
      dataFile.close();
   }
   // si no existe mostrar error
   else {
      Serial.println("No se ha podido abrir el fichero indicado.");
   }
}

void loop(){
}
```

Tal y como podemos ver en el código, la forma de leer datos del fichero es muy semejante a la manera en la que trabajábamos con los datos recibidos por conexiones serie: se obtiene si existe disponibilidad de datos para leer por medio de la llamada a `available()` y en caso de existir se leen mediante `read()`:

```
while (dataFile.available()) {
   Serial.write(dataFile.read());
}
```

Al ejecutar el *sketch* y si se ha grabado bien el fichero del ejemplo anterior, obtendremos en el monitor serie una salida semejante a:

```
Iniciando tarjeta...
Tarjeta inicializada.
Humedad:37.0 , Temperatura: 21.6
Humedad:34.9 , Temperatura: 21.8
Humedad:37.0 , Temperatura: 21.8
Humedad:36.5 , Temperatura: 21.6
Humedad:36.7 , Temperatura: 21.6
Humedad:36.8 , Temperatura: 21.0
```

Si no fue bien la grabación, el fichero no se habrá creado y la salida en el monitor serie será:

```
Iniciando tarjeta...
Tarjeta inicializada.
No se ha podido abrir el fichero indicado.
```

Una modificación útil para este ejemplo sería utilizar la librería Time para poder guardar el momento temporal exacto en el cual se ha producido la lectura.
En estos ejercicios se ha visto cómo comenzar a utilizar la tarjeta y a abrir el archivo para trabajar con él, pero también es posible realizar otras acciones sobre la tarjeta SD.
Del mismo modo que hemos sido capaces de crear ficheros, también podemos borrarlos a través de:

```
SD.remove(nombre_fichero)
```

que eliminará el fichero indicado por el parámetro `nombre_fichero` devolviendo `true` si la eliminación ha sido satisfactoria y `false` si ocurrió algún problema.
También podemos comprobar la existencia de un fichero o de un directorio mediante:

```
SD.exists(nombre_fichero)
```

que devolverá un booleano indicando la existencia del fichero o directorio informado en el parámetro `nombre_fichero`; este parámetro debe contener la ruta completa al fichero o directorio desde la raíz del sistema de archivos de la tarjeta.
En cuanto a los directorios, podemos también crear y borrar por medio de las llamadas:

```
SD.mkdir(nombre_directorio)
SD.rmdir(nombre_directorio)
```

Respecto a la creación mediante `mkdir()`, hay que comentar que es capaz de generar el directorio especificado y todos aquellos en la ruta que no existan, por ejemplo si utilizamos una tarjeta en la que no existe ningún fichero ni directorio y ejecutamos:

```
SD.mkdir("primer/segundo/tercer")
```

se creará el directorio `primer` en el raíz de la tarjeta, luego se creará `segundo` dentro de `primer` y por último `tercer` dentro de `segundo`, todo en una misma sentencia. Si todo va correcto, la función devuelve `true` y `false` se ha producido algún error.
En cuanto al borrado de los directorios, advertir que tan sólo se borrarán los directorios que se encuentren completamente vacíos y la función devolverá `true` en caso de que se haya podido borrar el directorio indicado.

Una vez se ha llamado a la función `SD.open()`, se obtiene una referencia a un objeto de la clase `File` sobre el cual se pueden realizar múltiples acciones, algunas de ellas ya vistas y otras no.
Por ejemplo entre las vistas están las de lectura (función `read()`) y escritura (funciones `print()`, `println()` y `write()`), la de cierre de archivo (función `close()`) y la de disponibilidad (función `available()`). Existen otras funciones que facilitan el trabajo con ficheros, como volcar la información pendiente de ser guardada en disco sin tener que cerrar el fichero:

```
fichero.flush()
```

donde `fichero` es la instancia devuelta por `SD.open()`.
Cuando se lee un archivo, existe un puntero que va indicando en qué posición del archivo se encuentra leyendo y cada vez que se llama a la función `read()` se actualiza el valor de este puntero avanzando por el archivo, del mismo modo que en el buffer de entrada serie podíamos leer un byte sin eliminar los datos también en este caso podemos leer un byte del fichero sin avanzar el puntero de lectura mediante la llamada a:

```
fichero.peek()
```

nuevamente `fichero` es la instancia al fichero abierto. Para conocer la posición en la que está el puntero de lectura debemos recurrir a la llamada:

```
fichero.position()
```

quien devuelve un entero largo sin signo con la posición en la que se encuentra; también es posible modificar dicha posición mediante:

```
fichero.seek(posición)
```

donde `posición` será la posición dentro del fichero a la que se quiere saltar, devolviendo `true` si todo ha ido de modo correcto y el puntero se ha actualizado a la posición indicada.
Para conocer si la instancia a fichero es realmente un fichero o un directorio se puede consultar:

```
fichero.isDirectory()
```

que devolverá `true` en caso de que la instancia sea un directorio y `false` si se tratara de un fichero.
Si lo que queremos es conocer el tamaño del fichero, valdría con realizar la llamada:

```
fichero.size()
```

que devuelve un entero largo sin signo con el tamaño en bytes de la instancia de archivo sobre la que se ejecuta.

14 Pantallas

En este capítulo aprenderá a:

- Conocer diferentes tipos de pantalla.
- Mostrar información en pantallas LCD.
- Mostrar información en pantallas TFT.
- Controlar pulsaciones en pantallas TFT táctiles.

Para proveer de información al usuario sobre lo que está pasando o en qué estado se encuentran los procesos que se están llevando a cabo dentro del microcontrolador, necesitamos de algún elemento capaz de transformar estos estados en señales comprensibles por el usuario.
Dependiendo de la naturaleza de los procesos y estados, podemos valernos simplemente de señales luminosas, como por ejemplo un led verde para cuando está encendido y rojo o nada para cuando está apagado; otra opción son las señales auditivas, por ejemplo un pitido indicando que se va a comenzar a mover un servomotor.
Pero existen otras situaciones en las que simples señales luminosas o auditivas no son capaces de proporcionar toda la información necesaria, por ejemplo para indicar una temperatura. En un primer caso podemos cubrir esta necesidad con displays de siete segmentos, pero sólo podríamos dar el número de temperatura; si además quisiéramos mostrar la palabra "temperatura" ya no nos valdría.
Para los casos en los que se necesitaba publicar datos complejos hemos ido utilizando varias soluciones como por ejemplo el monitor serie, pero claro está que en un circuito final no dispondremos de dicho monitor; otra opción anteriormente utilizada era comunicar los datos a un ordenador vía comunicación serie o Internet y que fuera el ordenador el encargado de presentar la información al usuario de modo ameno.
En este capítulo veremos cómo usar diferentes tipos pantallas, de modo que podamos tener un circuito autónomo capaz de mostrar información compleja al usuario sin depender de un ordenador ni de otros elementos externos al propio circuito.

Pantallas LCD

Las pantallas LCD (*Liquid Cristal Display*, pantallas de cristal líquido) son unos dispositivos planos que utilizan las propiedades de modulación de la luz de ciertos cristales líquidos para mostrar información.
Las primeras pantallas LCD disponibles en el mercado estaban compuestas de pequeños segmentos a modo de los display de 7 segmentos y aunque todavía hoy pueden encontrarse estos tipos de pantallas, lo normal es encontrarlas formadas por una matriz de pequeños puntos a modo de pixel que al ser iluminados son capaces de mostrar un rango mayor de formas que los 7 segmentos. Ambos tipos de pantalla utilizan la misma tecnología intrínsecamente, con la excepción de el número de elementos iluminados que entran en juego a la hora de mostrar cualquier carácter es distinto; puesto que para

mostrar el carácter en las pantallas de segmentos se utilizan menos elementos, pero su definición y versatilidad es menor al estar sujeta a las posibles formas que se puedan realizar con dichos segmentos; por otro lado las pantallas de matriz o bien pixel, son múltiples los elementos a iluminar pero son capaces incluso de mostrar gráficos; cuantos más píxeles tenga la pantalla, mejor resolución presentará la imagen. También existen pantallas con combinaciones de ambas estrategias.
Este tipo de pantallas son ampliamente utilizadas en los dispositivos electrónicos por múltiples razones:

- Bajo coste: Son más caras que las pantallas de 7 segmentos pero mucho más baratas que las TFT, son pantallas que para la funcionalidad que ofrecen tienen un precio contenido.
- Durabilidad: Suelen tener una vida larga; no sólo son resistentes en funcionamiento soportando infinidad de horas encendida, sino que también son muy resistentes a golpes y tensiones mecánicas (como todo... hasta cierto punto).
- Bajo consumo: Consumen poca energía lo que las hace especialmente atractivas para dispositivos portátiles, con funcionamiento a pilas o en pantallas que tengan que estar continuamente encendidas como pueden ser las de los relojes. En comparación con las pantallas de tubo de rayos catódicos (también conocidas como pantallas CRT) necesitan mucha menos potencia eléctrica para operar.
- Múltiples tamaños: Es fácil crear pantallas de múltiples tamaños, desde tan pequeñas como un reloj o un termómetro digital hasta las pantallas de grandes televisores.
- No sufren el efecto burn-in: Algunos tipos de pantallas, sobre todo las que usan algún tipo de fósforo para mostrar las imágenes, al estar mucho tiempo encendidas con una misma imagen, acaban quemando la superficie de proyección de modo que al apagar la pantalla, la imagen se sigue viendo; es el conocido como efecto *burn-in*. Aunque no sufren *burn-in* pueden llegar a generar un efecto similar denominado persistencia de imagen que se debe a que los cristales al estar mucho tiempo bajo el mismo voltaje acaban polarizándose y teniendo tendencia a quedarse en esa posición. Normalmente esta persistencia de imagen es temporal y acaba desapareciendo al cabo de un tiempo sin tener excitación eléctrica pero en ocasiones puede llegar a ser permanente; aunque es difícil que llegue a suceder esta situación es uno de los casos de malfuncionamiento que no suele estar cubierto por las garantías.

Como siempre, las desventajas dependerán del modelo de pantalla que utilicemos y cómo no del dinero que podamos gastar, pero entre las mayores desventajas podemos nombrar:

- Bajo ángulo de visión: La imagen de este tipo de pantallas es totalmente nítida cuando se mira de modo perpendicular a ella y va perdiendo definición de manera muy rápida cuando nos separamos de esta posición.
- Problemas para mostrar imágenes en movimiento: Al mostrar imágenes que se mueven rápidamente causa un efecto niebla (*motion blur*).
- Problemas de uso a bajas temperaturas: Puede ocurrir que a temperaturas bajo cero lleguen incluso a dejar de funcionar estas pantallas.
- Pixel muerto: Se trata de uno de los temidos problemas de este tipo de pantallas. Sucede cuando uno de los segmentos o pixeles deja de excitarse correctamente al recibir corriente eléctrica, mostrándose siempre como desactivado.

Los cristales líquidos en sí no emiten luz, sino que lo que hacen es bloquearla. En un estado de no excitación, es decir sin corriente eléctrica, los cristales dejan pasar toda la luz y es en el momento en el que se les aplica voltaje cuando se reordenan para no dejar pasar la luz de ciertas longitudes de onda. Lo que se hace en este tipo de pantallas es introducir una fuente de luz tras los cristales líquidos y cuando están estos excitados por un voltaje eléctrico, bloquean esta luz posterior dando lugar a las imágenes que podemos ver.
Existen muchas tecnologías para realizar pantallas LCD, incluso algunas capaces de retener la imagen y seguir mostrándola aún en ausencia de electricidad. Una de las tecnologías más difundidas consiste en que cada pixel de la pantalla LCD, está formado por una capa de moléculas alineadas entre dos electrodos transparentes y dos filtros polarizantes posicionados normalmente de manera que los ejes de polarización sean perpendiculares entre ellos, así la luz que deja pasar el primero la bloquea el segundo. Sin aplicar tensión a la pantalla, la orientación de las moléculas de cristal líquido viene determinada por el alineamiento sobre las superficies de los electrodos que son tratados de modo que las moléculas en contacto con ellos tengan un alineamiento perpendicular entre ellas, colocándose las moléculas centrales en estructuras helicoidales como se muestra en la figura 14.1. La luz al incidir sobre los pixeles recibe una distorsión que hace que los veamos de un color grisáceo. Si se aplica una diferencia de potencial suficiente, las moléculas del centro de la capa abandonan las estructuras helicoidales y la luz atraviesa la capa del cristal sin tener distorsión en su polarización e incide de modo perpendicular sobre el segundo filtro que bloquea la luz completamente y aparece como

negro. Dependiendo de la diferencia de potencial aplicada sobre la capa de cristal líquido, se controla la cantidad de luz capaz de cruzarla y así se obtienen diferentes escalas de grises. Este control se realiza sobre cada pixel del conjunto de la pantalla; para controlar los datos a mostrar y la cantidad de voltaje a aplicar sobre cada pixel, las pantallas vienen con una serie de circuitos adicionales denominados drivers LCD.

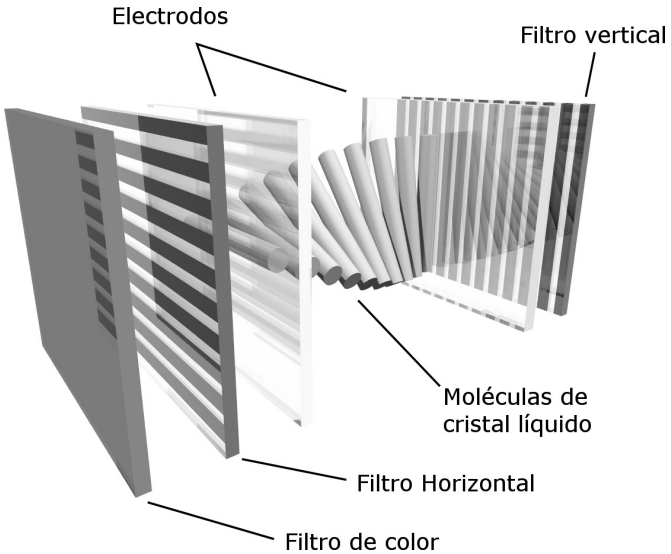


Figura 14.1. Esquema de subpixel de pantalla LCD.

Las matrices de píxeles pueden ser de dos tipos: activas y pasivas.
Las matrices pasivas fueron las primeras en utilizarse y en ellas cada pixel debe encargarse de mantener su estado sin necesidad de circuitos externos hasta que se vuelva a refrescar. Estas pantallas dividen su alimentación por filas (señal *select*, de selección) y columnas (señal de *video*). Mediante la señal *select* se selecciona la fila sobre la que operar y mediante la señal *video* nos encargaríamos de ir recorriendo las columnas e ir seleccionando cada pixel de modo individual e ir aplicando la tensión necesaria en caso de querer que el pixel se ilumine. Para poder iluminar una pantalla de m x n píxeles necesitaremos m + n conectores para ser capaces de direccionar todos los píxeles.
Las matrices activas son las que podemos encontrar hoy en día en la mayoría de las pantallas planas. Usada por primera vez en 1989, esta tecnología y variaciones de ella que veremos más adelante se usan de modo profuso hoy en día. En esta tecnología, cada pixel está unido a un transistor quien de manera activa mantiene el estado del pixel mientras se accede a los otros

píxeles; siendo esta la diferencia con las matrices pasivas que no tenían circuitos externos para mantener el estado. El número de conexiones para poder direccionar todos los píxeles y así iluminarlos es igual que en el caso de las matrices pasivas m + n. Una de las pantallas de cristal líquido con matriz activa más utilizadas hoy en día es la conocida TFT (Thin-film-transistor liquid-crystal display, pantalla de cristal líquido de transistor de capa fina) que veremos más adelante.

Figura 14.2. Pantalla LCD 16x2.

En Arduino podemos encontrar múltiples modelos de pantallas LCD aunque quizá las más conocidas sean las denominadas 16x2, por tener 16 bloques de matrices por línea y 2 líneas de matrices; una compatible con el controlador Hitachi (muy fáciles de encontrar en el mercado ya que casi todas lo son) será la que utilicemos para los ejemplos. Lo primero que se debe saber de este tipo de pantallas, es que necesitan múltiples conexiones para que funcione correctamente, por lo que se debe poner mucha atención a la hora de realizar el cableado del circuito. Los pines necesarios a cablear son:

- RS (*Register Select*, Selección de Registro): Controla en qué lugar de la memoria de la pantalla se va a escribir. Puede seleccionar el registro de datos, que controla lo que se muestra en pantalla o el registro de instrucciones que es donde se guarda la instrucción siguiente a procesar por parte del controlador de pantalla.
- R/W (*Read/Write*, Lectura/Escritura): Selecciona si se va a leer o a escribir.
- E (*Enable*/Habilita): Habilita la escritura en los registros.
- V0: Pin de contraste de pantalla.
- D0-D7: Pines de datos, serán usados para escribir en los registros o leer de ellos.
- Pines de tensión para controlador y led de iluminación.

Vamos a detallar los terminales de conexión de la tarjeta Arduino con la pantalla ya que es posible que el gráfico que acompaña al *sketch* pueda resultar algo lioso por el número de conexiones o que pueda tenerse una pantalla con los pines en una disposición distinta a la mostrada.

Pin Pantalla	Pin Arduino
VSS	GND
VDD	5V
RS	7
R/W	GND
E	8
D4	9
D5	10
D6	11
D7	12
A(led +)	5V
K(led -)	GND

Además utilizaremos el pin V0 para conectar con un potenciómetro y permitirá controlar la intensidad de la imagen en la pantalla.
Podremos observar que sólo se han conectado 4 de los 8 terminales de datos; esto es porque las estas pantallas pueden trabajar en modo 4 bits o modo 8 bits siendo necesarios distinto número de conexiones para cada uno de ellos; puesto que trabajando con 4 bits es posible realizar casi todo el trabajo de mostrar textos en pantalla, trabajaremos de esta forma y sólo necesitaremos conectar los pines D4-D7.

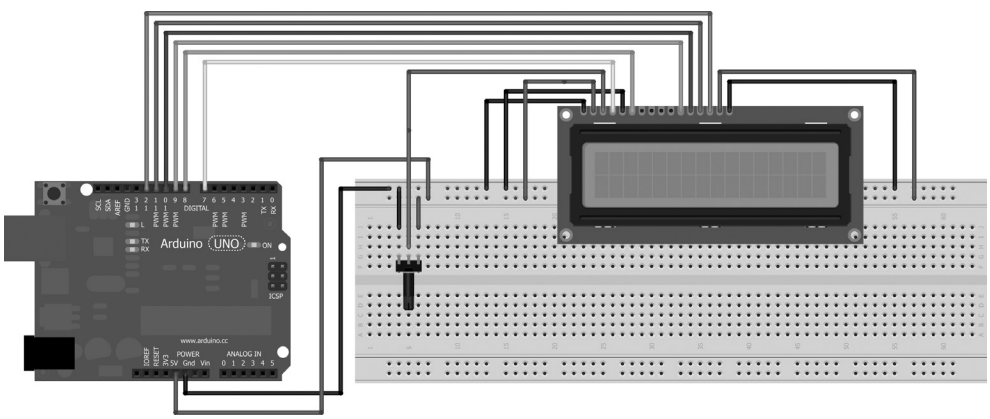


Figura 14.3. Conexiones para circuito con pantalla LCD.

Cuando se quiere mostrar algo por pantalla, debe llevarse al registro de datos la imagen del dato que se quiere mostrar e indicar después al registro de instrucciones que se encargue de mostrarlo. Para trabajar con las pantallas LCD, Arduino incorpora una librería de ayuda que facilitará estas tareas y se puede hacer disponible al *sketch* desde el menú Sketch>Importar librería...> LiquidCrystal o añadiendo al código la línea:

```
#include <LiquidCrystal.h>
```

Para crear la instancia de la clase `LiquidCrystal`, se dispone de varias opciones pero dado que no vamos a trabajar en 8 bits ni vamos a usar el pin R/W, utilizaremos:

```
LiquidCrystal(pinRS, habilitado, pinD4, pinD5, pinD6, pinD7)
```

Donde los parámetros corresponden con los pines descritos anteriormente. Ya teniendo la instancia, podemos acceder a la pantalla a través de las funciones de la clase `LiquidCrystal`, por ejemplo para escribir un texto sería mediante la función `print()`, de modo semejante a como se ha hecho con otros componentes. Para crear el "Hola Mundo!" sería tan sencillo como:

```
#include <LiquidCrystal.h>

LiquidCrystal myLCD(7, 8, 9, 10, 11 , 12);

void setup(){
   myLCD.print("Hola Mundo!");
}

void loop() {}
```

Figura 14.4. Hola Mundo en LCD.

Pero está claro que mostrar simplemente este mensaje se queda muy corto, queremos algo más; veamos que nos ofrece la libraría LiquidCrystal.
Aunque en este primer ejemplo no se ha hecho por ser un ejemplo bastante sencillo, lo primero que se debe hacer en la función `setup()` es iniciar la referencia a la pantalla LCD con la información sobre cuantas filas y cuantas columnas tiene para poder luego utilizar aspectos como la dirección en la que se escriben los textos o el *scrolling* (desplazamientos) y que la librería sepa calcular la posición en la que le toca dibujar el carácter.
La inicialización se realiza:

```
lcd.begin(columnas,filas)
```

informando el número de columnas y filas de la pantalla a utilizar. Recalcar que el primer parámetro son las columnas y no las filas como normalmente estamos acostumbrados.
Cuando se van mostrando caracteres en pantalla, funciona del mismo modo que cuando escribimos en el ordenador (salvando las distancias); un cursor se va moviendo por la pantalla y cada vez que imprime un carácter, pasa a la siguiente posición.
La visibilidad del cursor se controla mediante las funciones:

```
myLCD.cursor()
```

para mostrarlo sobre el carácter en el que está posicionado y sobre el que se escribiría en caso de introducir algo nuevo y:

```
myLCD.noCursor()
```

para ocultarlo. El cursor se puede mostrar en estado fijo o parpadeando; estas opciones se controlan mediante:

```
myLCD.blink()
```

para que se muestre parpadeando y para que se muestre estático utilizaríamos:

```
myLCD.noBlink()
```

Puede que en algunas pantallas no se muestre nunca el cursor o que no parpadee, y es que dependiendo del modelo de pantalla puede darse el caso que el controlador no soporte alguna de estas opciones.
Para mover el cursor a una determinada posición dentro de la pantalla, se dispone de la función:

```
myLCD.setCursor(columna,fila)
```

en la cual se le puede indicar mediante los parámetros `columna` y `fila` la posición donde se quiere colocar para escribir; los índices tanto de las filas como de las columnas comienzan en 0.

Si se quiere tener en la posición inicial (0,0), es decir en el carácter superior izquierdo se puede llamar a la función:

```
myLCD.home()
```

Si se quiere limpiar la pantalla, entonces utilizaremos la función:

```
myLCD.clear()
```

que además de limpiar la pantalla coloca el cursor en la posición inicial, en el carácter superior izquierdo (0,0). Si no se desea que el cursor vaya a la posición inicial, lo que se debe hacer es una combinación de funciones; se tiene que utilizar la función `myLCD.clear()` y después utilizar `myLCD.setCursor(columna, fila)` para posicionar donde se quiera. En español, al igual que en la mayoría de idiomas occidentales, la escritura se realiza de izquierda a derecha pero para representar algunos idiomas o simplemente para hacer efectos con el texto, se dispone de la posibilidad de indicar la dirección hacia donde queremos escribir el texto. Las dos opciones posibles son:

```
myLCD.leftToRight()
myLCD.rightToLeft()
```

siendo lógicamente la primera de ellas para escribir de izquierda a derecha y la segunda para hacerlo de derecha a izquierda.
En el caso de que queramos utilizar un carácter extraño o simplemente alguno que no exista dentro del código ASCII, existe la función:

```
myLCD.createChar(número, datos)
```

en este caso, el parámetro `número` indica en qué posición de la memoria se quiere guardar el carácter creado; se disponen de 8 posiciones numeradas de 0 a 7 para guardar caracteres creados por nosotros. Más adelante a la hora de imprimir el carácter, debe indicarse el número de posición de memoria en el que se ha guardado el carácter a imprimir. En cuanto al parámetro `datos`, se trata del dibujo que representa al carácter a imprimir; estos datos vienen dados por un array de 8 elementos de tipo byte de los cuales solamente se informan los 5 bits de menor peso, poniendo a 1 los bits que se deben encender para formar el gráfico que queramos mostrar como carácter y a 0 los que no. Son 8 elementos del array porque son los píxeles que tiene de alto el carácter en la pantalla, del mismo modo, simplemente informamos los 5 bits de menor peso porque el ancho del carácter en pantalla son 5 píxeles. Más adelante se utilizará la función `write()` para escribir este carácter en pantalla a partir de la posición en la que se ha almacenado.
Para que el cursor no se desplace y sea el texto quien lo haga, podemos hacer que el siguiente carácter que se dibuje "empuje" al resto de ellos para hacerse hueco; si la dirección de escritura del texto está configurada de izquierda

a derecha, entonces el desplazamiento se realizará hacia la izquierda y si está configurado de derecha a izquierda el desplazamiento se hará hacia la derecha. Para habilitar esta manera de trabajar se utiliza la llamada:

```
myLCD.autoscroll()
```

y cuando se desee no seguir trabajando más de este modo, se deshabilitará utilizando:

```
myLCD.noAutoscroll()
```

También podemos realizar manualmente los desplazamientos del texto a izquierda y derecha usando respectivamente:

```
myLCD.scrollDisplayLeft()
myLCD.scrollDisplayRight()
```

Por último cuando no queramos trabajar más con la pantalla, para ahorrar energía podemos apagarla mediante:

```
myLCD.noDisplay()
```

y volver a encenderla utilizando:

```
myLCD.display()
```

Ahora que ya sabemos las posibilidades de la librería, vamos a hacer un pequeño *sketch* en el que mostraremos el texto "Hola don Pepito..." en la primera línea y el texto "hola don José" en la segunda. Lo primero que tenemos que ver es que el carácter "é" no existe en ASCII, por lo que tendremos que fabricarlo nosotros. Además para ver funcionalidad explicada, los puntos suspensivos de la primera frase realizarán un desplazamiento automático a la izquierda y para limpiar los textos de pantalla (que no el buffer), desplazaremos los textos hacia la derecha.
Para crear el carácter "e" necesitamos un array de 8 posiciones con bytes cuyos 5 bits de menor peso contengan a 1 los caracteres que se quieren imprimir. Esto lo conseguimos mediante:

```
byte e[8] = {
   B00010,
   B00100,
   B01110,
   B10001,
   B11111,
   B10000,
   B01110,
   B00000
};
```

Se que es un momento un poco Matrix, pero realmente si nos fijamos en la posición de los 1 en el array, podremos ver el carácter "é".

Durante la inicialización del *sketch*, indicaremos las columnas y filas del la pantalla y crearemos el carácter "é" en la posición 0 de las 8 posiciones disponibles para guardar nuestros gráficos.

```
myLCD.begin(16,2);
myLCD.createChar(0, e);
```

En el cuerpo del *sketch* será donde vayamos escribiendo los textos a mostrar en pantalla. La escritura de los textos se realiza del mismo modo que se ha hecho en el ejemplo anterior, usando la función `print()`. Dado que los puntos suspensivos deben desplazar el texto a la izquierda, activamos el desplazamiento automático justo antes del momento de pintarlos:

```
// scroll automático
myLCD.autoscroll();
for (byte i = 0; i < 3; i++) {
   myLCD.print(".");
   delay(500);
}
// scroll automático apagado
myLCD.noAutoscroll();
```

Para escribir la segunda frase, se debe tener en cuenta que al pintar los puntos suspensivos, nos hemos desplazado 3 posiciones a la izquierda, por lo que si se comienza a escribir en la primera columna, estaríamos escribiendo fuera de pantalla ya que la primera columna que se ve ahora es la cuarta; esto lo solucionamos posicionando el cursor en esta columna, es decir en la columna con índice 3:

```
myLCD.setCursor(3,1);
```

Advertencia:

Cuando se trabaja con la librería LiquidCrystal, las funciones que trabajan con filas y columnas toman como primer parámetro la columna en lugar de la fila como estamos normalmente acostumbrados.

Para poder escribir la frase "Hola don José", escribiremos normalmente hasta el carácter "é" y éste será impreso recurriendo al gráfico creado por nosotros.

```
myLCD.print("Hola don Jos");
myLCD.write(byte(0));
```

Por último desplazaremos todo el texto a la derecha para sacarlo de pantalla.

```
for (byte i = 0; i < 19; i++){
   myLCD.scrollDisplayRight();
   delay(300);
}
```

No son sólo 16 posiciones las que tenemos que desplazar para sacar todo el texto de pantalla, ya que previamente habíamos desplazado 3 a la izquierda, por eso el bucle es de 19 posiciones.
Poniendo todo en un *sketch* quedaría:

```
#include <LiquidCrystal.h>

LiquidCrystal myLCD(7, 8, 9, 10, 11 , 12);

byte e[8] = {
   B00010,
   B00100,
   B01110,
   B10001,
   B11111,
   B10000,
   B01110,
   B00000
};

void setup() {
   myLCD.begin(16,2);
   // creación del caracter é
   myLCD.createChar(0, e);
}

void loop() {
   // colocamos cursor en (0,0):
   myLCD.setCursor(0, 0);
   // primera frase
   myLCD.print("Hola don Pepito");
   delay(1000);
   // scroll automático
   myLCD.autoscroll();
   for (byte i = 0; i < 3; i++) {
      myLCD.print(".");
      delay(500);
   }
   // scroll automático apagado
   myLCD.noAutoscroll();

   // colocamos el cursor en (3,1):
   myLCD.setCursor(3,1);
   myLCD.print("Hola don Jos");
   myLCD.write(byte(0));
   delay(1000);
   // movemos todo el texto a la derecha
   for (byte i = 0; i < 19; i++){
      myLCD.scrollDisplayRight();
      delay(300);
   }
   delay(1500);
   myLCD.clear();
}
```

Figura 14.5. Texto en LCD.

Para dar una mayor utilidad a la pantalla LCD, vamos a variar un montaje realizado anteriormente.
En el capítulo 8, ya veíamos el sensor LM35 para poder realizar lecturas de temperatura, que se mostraban en el monitor serie; utilizaremos ahora la pantalla LCD para mostrar el resultado de estas lecturas.
El circuito a montar es una mezcla del visto en el capítulo 8 y el del ejemplo anterior, donde para su construcción necesitaremos simplemente la pantalla LCD y el sensor LM35 o uno semejante.

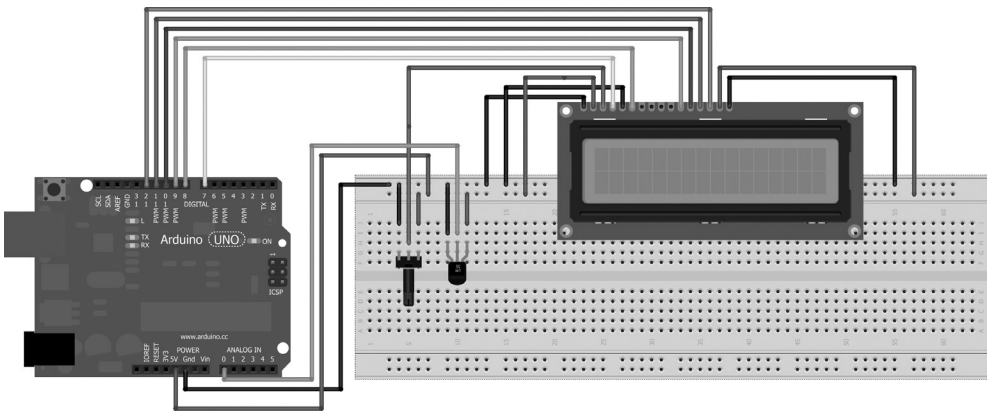


Figura 14.6. Circuito de lectura de temperaturas.

En el *sketch* se deben ir realizando lecturas al puerto A0 que será donde esté conectado el sensor y una vez recuperado el dato mostrarlo en pantalla. En la primera línea se mostrará el texto "Temperatura:" y en la segunda la temperatura leída seguido de "ºC". Nuevamente el carácter "º" no existe, por lo que deberemos crearlo como se hizo en el ejemplo anterior.
Para dar algo de vistosidad a lo mostrado en la pantalla, escribiremos todo para que no se vea y luego desplazaremos el texto hacia la izquierda de modo que irá apareciendo por la derecha y desplazándose poco a poco de derecha a izquierda. Para conseguir este efecto se debe escribir en la columna 16 (más allá de lo mostrado actualmente en pantalla, si son 16 columnas, la última mostrada es la de índice 15). Una vez escrito el texto, simplemente debemos desplazarlo a la izquierda mediante `myLCD.scrollDisplayLeft()`.

```
#include <LiquidCrystal.h>

LiquidCrystal myLCD(7, 8, 9, 10, 11 , 12);

float tempC; // temperatura en celsius
int analogValue; // valor leído en el puerto
int sensorPin = A0;
// carácter °
byte grados[8] = {
   B00000,
   B00100,
   B01010,
   B00100,
   B00000,
   B00000,
   B00000,
   B00000
};

void setup() {
   myLCD.begin(16,2);
   analogReference(INTERNAL);
   // creación del carácter
   myLCD.createChar(0, grados);
}

void loop() {
   // colocamos cursor en (16,0):
   myLCD.setCursor(16, 0);

   myLCD.print("Temperatura:");
   analogValue = analogRead(sensorPin);
   tempC = analogValue / 9.31;   // se divide el valor leido por el número
                                 // calculado en el capítulo

   // colocamos el cursor en (16,1):
   myLCD.setCursor(16,1);
   myLCD.print(tempC);
```

```
   myLCD.print(" ");
   myLCD.write(byte(0));
   myLCD.print("C");
   delay(1000);
   for (byte i = 0; i < 16; i++){
      myLCD.scrollDisplayLeft();
      delay(300);
   }

   delay(1500);
   myLCD.clear();
}
```

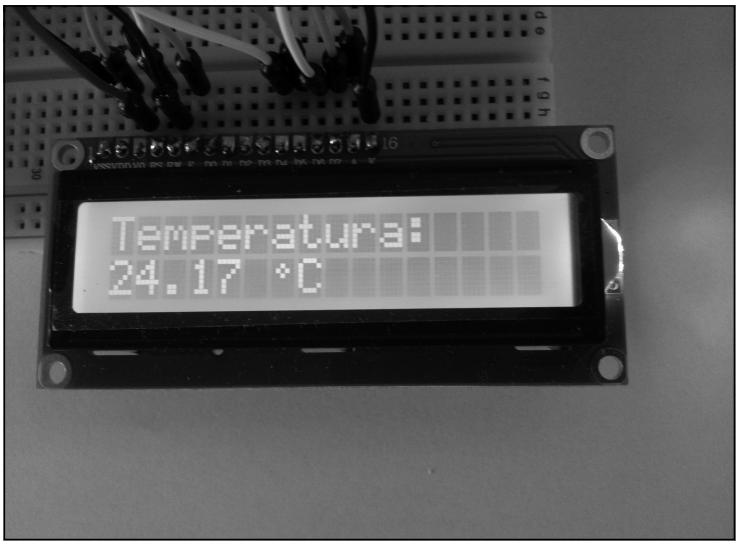


Figura 14.7. Temperaturas en LCD.

Para mostrar pequeños textos o informaciones estas pantallas son suficientemente válidas, pero cuando se quiere mostrar textos complejos o algún tipo de imagen debemos recurrir a otro tipo de pantallas como las TFT.

Pantallas TFT

Las pantallas TFT (Thin-film-transistor liquid-crystal display, pantalla de cristal líquido de transistor de capa fina) son un tipo de pantallas de cristal líquido que se encuentra muy extendido hoy en día en los aparatos electrónicos. Casi todas las pantallas de televisión, monitores, pantallas de móviles, tabletas o portátiles son de tipo TFT o alguna de sus variantes.

Las pantallas TFT son unas pantallas de matriz activa que frente a las tradicionales de cristal líquido (o simplemente LCD) tienen una mayor calidad de imagen, más contraste y un ángulo de visión mucho mayor.
La manera en la que los transistores encargados de mantener el estado de iluminación del pixel se fabrican, es lo que le da esta mayor calidad de imagen a las pantallas TFT. Los transistores normalmente se generan a partir de cristal de silicio en una oblea, pero en su lugar, estos transistores se fabrican depositando el silicio sobre paneles de cristal creando una capa muy fina. Además los transistores así fabricados ocupan una porción muy pequeña del tamaño total del pixel, por lo que no "molestan" en el paso de la luz y pueden dar así mayor brillo.
Hay un tipo de pantallas TFT que están muy de moda últimamente debido a los terminales móviles (teléfonos y tabletas principalmente) que son las pantallas IPS (*in-Plane Switching*, cambio en plano). Estas pantallas son una variación más de las LCD TFT, donde las moléculas de cristal líquido se mueven de modo paralelo a la pantalla en lugar de perpendicular a ella como hacen en las TFT, lo cual produce un mayor ángulo de visión. Inicialmente esta tecnología ofrecía muy buen ángulo de visión y gran nitidez en los colores, pero tenía tiempos de respuesta muy altos; actualmente variaciones sobre esta tecnología han conseguido mejorar los tiempos de respuesta de la imagen, como por ejemplo las pantallas SuperIPS y es por ello que este tipo de pantallas están siendo adoptadas en monitores de alta gama y dispositivos móviles.

Figura 14.8. Pantalla TFT.

Para realizar circuitos con Arduino podemos optar por comprar una pantalla TFT cualquiera o decantarnos por algunas de las que hay en el mercado que se encuentran preparadas para trabajar con Arduino; la diferencia estriba en que las genéricas se deben cablear a mano, mientras que las preparadas para trabajar con Arduino suelen venir de modo que los terminales vienen posicionados de manera que podemos pincharla directamente sobre nuestra tarjeta.

En caso de decantarse por una pantalla para utilizar pinchada directamente sobre la tarjeta Arduino, hay que fijarse para qué modelo está preparada, ya que algunas son específicas para Arduino Mega.
A modo orientativo esta sería la configuración de conexiones a realizar si se opta por una pantalla suelta:

Terminal Pantalla	Terminal Arduino
BD0 a BD7	D0 a D7
RSET	A2
CS	A3
RW	A4
RS	A5
RD	5V o 3.3V
GND	GND
VCC	5V

No obstante hay que recordar que siempre se debe leer la información del fabricante para conocer las tensiones de trabajo y otras indicaciones que puedan resultar útiles para la operatividad de los elementos.
Programar las pantallas TFT directamente es una tarea muy compleja y tediosa, pero existe librerías para facilitar la tarea. Podemos optar por utilizar la librería TFT presente a partir de la versión 1.0.5 de Arduino o bien una librería llamada UTFT que será la que utilicemos por ser más completa; esta librería se puede descargar de manera gratuita desde la Web de su creador (`http://henningkarlsen.com/electronics/library.php?id=51`).
Para instalar la librería, se debe copiar en el directorio `libraries` dentro del directorio de instalación del entorno de programación Arduino mientras no estemos ejecutando el entorno (o reiniciarlo después, de lo contrario dará error durante la compilación).
Son muchos los modelos, tamaños y resoluciones de pantallas; por ejemplo podemos tener pantallas de 2.4 pulgadas en blanco y negro con una resolución de 320 x 240 o de 5 pulgadas con una resolución de 800 x 480 táctiles y con lector de memorias SD. Dado todo este abanico de posibilidades, la librería UTFT no da soporte a cualquier pantalla, da a muchas, pero hay algunas en el mercado que no están soportadas. Dentro de la documentación de la librería se detallan las pantallas y controladores soportados por ésta.

Para trabajar con esta librería lo primero que se debe hacer es incluirlas en el programa mediante:

```
#include "UTFT.h"
```

Una vez incluida debemos crear una instancia de la clase `UTFT` que representará a la pantalla a controlar mediante el constructor:

```
UTFT nombre_variable(modelo,pinRS,pinWR,pinCS,pinRST);
```

Donde `modelo` indica el módulo de la pantalla (hay que mirar el módulo que corresponde a la pantalla que vayamos a usar), `pinRS` define el pin de selector de registro, `pinWR` el pin de escritura, `pinCS` el pin de selección de chip y `pinRST` el pin de reiniciar. Los pines analógicos serán utilizados como pines digitales, y así los informaremos durante la creación de la instancia. Los pines analógicos son numerados a partir de los pines digitales, es decir en la placa Arduino Uno los pines digitales terminan en el número 13, pues el pin analógico A0 sería el pin digital 14 y así sucesivamente:

Pin Analógico Uno	Pin Digital Uno
A0	14
A1	15
A2	16
A3	17
A4	18
A5	19

Para la tarjeta Arduino Mega funcionaría de modo análogo. Teniendo esto en cuenta, tendríamos la inicialización:

```
UTFT myTFT(ILI9325C,19,18,17,16); // para Uno
UTFT myTFT(ITDB32S, 38,39,40,41); // para Mega
```

Nota:

Cada pantalla tiene un modelo de controlador que tiene que ser informado a la hora de crear la instancia de la clase UTFT.

Durante la función de `setup()` indicaremos la orientación de la pantalla y la limpiaremos mediante las llamadas a las funciones:

```
myTFT.InitLCD(orientación);
myTFT.clrScr();
```

Donde `orientación` serían las constantes `LANDSCAPE` o bien `PORTRAIT` dependiendo si se desea trabajar con la pantalla en horizontal o en vertical respectivamente. En caso de no informarse `orientación` con alguna de las dos constantes, se trabajaría en horizontal. Ya estaríamos en disposición de trabajar y pintar sobre la pantalla.
Para poder escribir texto en pantalla, UTFT pone a nuestra disposición la función `print()` que tiene por parámetros el texto a imprimir y las coordenadas (X,Y) donde realizarlo tomando como eje de coordenadas la esquina superior izquierda.
Antes de poder imprimir el texto es necesario seleccionar la fuente que se usará para ello. Aunque es posible añadir nuestras propias fuentes, para agilizar el trabajo, UTFT dispone de tres fuentes que podemos utilizar directamente:

- SmallFont: fuente de tamaño pequeño.
- BigFont: fuente de tamaño grande.
- SevenSegNumFont: fuente imitando los displays de 7 segmentos.

Para poder utilizarlas es indispensable añadir las líneas:

```
extern uint8_t SmallFont[];
extern uint8_t BigFont[];
extern uint8_t SevenSegNumFont[];
```

Y más adelante seleccionar aquellas que nos interese por medio de la función `setFont()`:

```
myGLCD.setFont(SmallFont);
```

Con esto ya podríamos hacer nuestro primer programa sobre una pantalla TFT, que será un "Hola mundo!":

```
#include "UTFT.h"

extern uint8_t BigFont[];

// cambiar dependiendo de la pantalla y tarjeta Arduino
UTFT myTFT(ILI9325C,19,18,17,16); // para Uno

void setup(){
   myTFT.InitLCD(LANDSCAPE);
   myTFT.clrScr();
   myTFT.setFont(BigFont);
}

void loop(){
   myTFT.print("Hola Mundo!", 0, 0);
   delay (1000);
   myTFT.clrScr();
   delay (1000);
}
```

Figura 14.9. Hola Mundo en TFT.

El programa no es gran cosa, simplemente muestra el texto "Hola Mundo!" en la esquina superior izquierda de manera intermitente a intervalos de un segundo, pero tampoco son muchas líneas de código.
Mediante las funciones `setColor()` y `setBackColor()` podemos controlar tanto el color de la letra como el color del fondo del texto. Su sintaxis es muy parecida:

```
myTFT.setColor (r,g,b);
myTFT.setBackColor (r,g,b);
```

Donde los parámetros corresponden a los valores RGB (Rojo Verde y Azul) que queremos dar al color; estos valores van de 0 a 255. En lugar de los tres parámetros ya vistos, esta función también admite alguna de las constantes de colores definidas en la librería como `VGA_FUCHSIA` o bien `VGA_BLUE`, estas constantes se pueden encontrar en el fichero `UTFT.h` dentro del directorio de la propia librería.
Pero antes de ponernos a jugar con los colores para ver el resultado, deberíamos centrar el texto.

UTFT dispone para ello de ciertas constantes que nos pueden ayudar en este trabajo:

- LEFT: Indica la coordenada para que el texto quede alineado a la izquierda de la pantalla.
- RIGHT: Indica la coordenada para que el texto quede alineado a la derecha de la pantalla.
- CENTER: Indica la coordenada para que el texto quede centrado.

En este ejemplo calcularemos números aleatorios para cada una de las componentes RGB de los colores del texto y del fondo de éste y mostraremos de nuevo el "Hola Mundo!" pero esta vez desplazado 100 pixeles del borde superior y centrado horizontalmente:

```
#include "UTFT.h"
extern uint8_t BigFont[];

// cambiar dependiendo de la pantalla y tarjeta Arduino
UTFT myTFT(ILI9325C,19,18,17,16); // para Uno

void setup(){
   myTFT.InitLCD(LANDSCAPE);
   myTFT.clrScr();
   myTFT.setFont(BigFont);
   randomSeed(analogRead(0));
}

void loop(){
   myTFT.setColor(random(0,255), random(0,255), random(0,255));
   myTFT.setBackColor(random(0,255), random(0,255), random(0,255));
   myTFT.print("Hola Mundo!", CENTER, 100);
   delay (1000);
   myTFT.clrScr();
   delay (1000);
}
```

Para dar más plasticidad a los textos, existe la posibilidad de indicar el grado de rotación con el que se quiere mostrar el texto, para ello vale con añadir como parámetro a la función `print()`, después de la coordenada Y, los grados que se quiere girar. Para hacer un *sketch* que muestre el texto "Hola mundo" centrado en la pantalla y que cada 15 grados lo imprima a modo de flor tomando como centro de rotación la 'H' de "Hola", debemos calcular previamente el centro de la pantalla. Si estaba pensando utilizar las coordenadas `CENTER`, no sirven ya que se calculan teniendo en cuenta el tamaño del texto para que quede centrado y lo que se pretende es que gire sobre la H; además el autor de la librería desaconseja el uso de `RIGHT` y `CENTER` cuando se utilizan rotaciones por no calcularse de modo correcto.

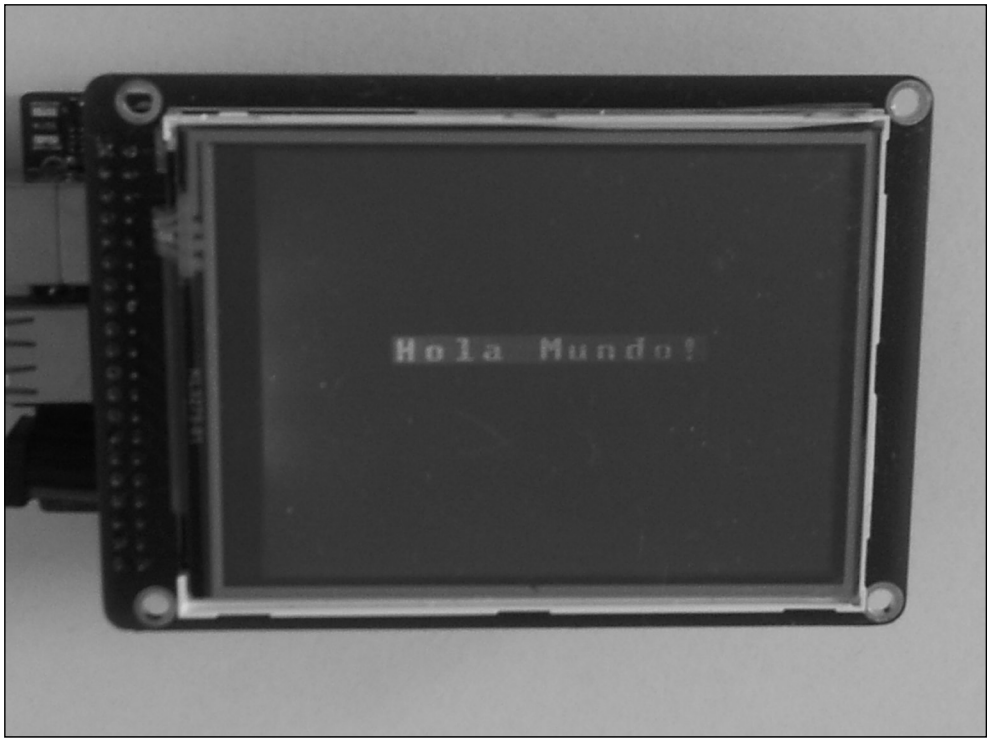


Figura 14.10. Hola Mundo con colores en TFT.

Para poder conocer el tamaño de la pantalla, existen las funciones:

```
myTFT.getDisplayXSize()
myTFT.getDisplayYSize()
```

Para calcular el centro de la pantalla simplemente se debe recoger lo obtenido por estas funciones y dividirlo entre dos.

```
#include "UTFT.h"

extern uint8_t SmallFont[];
// cambiar dependiendo de la pantalla y tarjeta Arduino
UTFT myTFT(ILI9325C,19,18,17,16); // para Uno
int xCenter;
int yCenter;

void setup(){
   myTFT.InitLCD(LANDSCAPE);
   myTFT.clrScr();
   myTFT.setColor(0, 0, 255);
   myTFT.setBackColor(0,0,0);
   myTFT.setFont(SmallFont);
   xCenter =  myTFT.getDisplayXSize() /2;
```

```
   yCenter =  myTFT.getDisplayYSize() /2;
}

void loop(){
   for (int i = 0; i < 360; i += 15){
      myTFT.print("Hola Mundo", xCenter, yCenter, i);
      delay (500);
   }
   myTFT.clrScr();
   delay (1000);
}
```

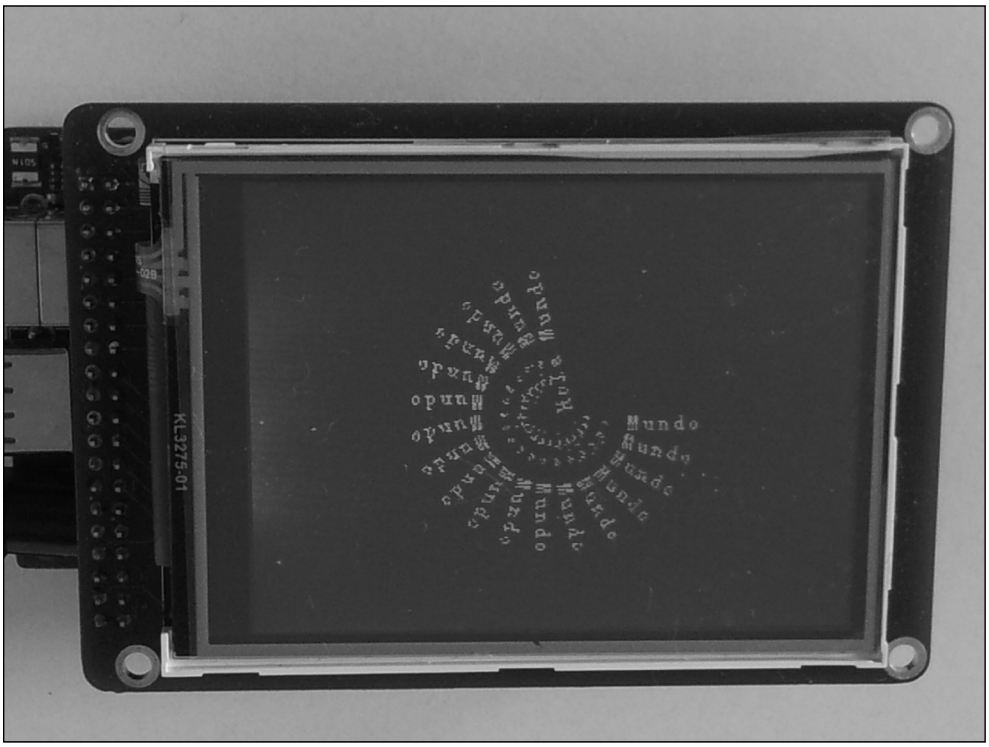


Figura 14.11. Hola Mundo con giro en TFT.

Además de la función para escribir texto en pantalla, existen dos funciones destinadas a escribir números:

```
myTFT.printNumI(num,x,y,tamaño,relleno);
myTFT.printNumF(num,dec,x,y,divisor,tamaño,relleno);
```

La función `printNumI()` va dirigida a números de tipo `int` mientras que la función `printNumF()` va dirigida a tipo `float`. El parámetro `num` se refiere al número a escribir, `x` e `y` son las coordenadas XY donde realizar la escritura,

`tamaño` es el tamaño mínimo de caracteres que debe ocupar el número, `relleno` sirve para indicar el carácter con el que rellenar para que el número ocupe el tamaño mínimo indicado (por defecto es el espacio en blanco), `dec` indica el número de caracteres para la parte fraccionaria y `divisor` es un carácter que se usará para separar la parte entera de la decimal. Los parámetros `divisor`, `tamaño` y `relleno` son opcionales.
De igual modo, si nos sentimos cómodos utilizando la función `print()`, no hay ningún problema en utilizarla con números.
Además de poder escribir texto, también es posible realizar figuras geométricas sobre la pantalla; para ello se cuenta con un grupo de funciones que permiten dibujar un pixeles, líneas, rectángulos (con bordes redondeados o no) y circunferencias.
En el siguiente ejemplo veremos cómo se hace para dibujar rectángulos y circunferencias, dejando los pixeles y líneas para más adelante.
Para dibujar un rectángulo necesitaremos las coordenadas de su vértice superior izquierdo y de su vértice inferior derecho y su uso es:

```
myTFT.drawRect(x1,y1,x2,y2);
```

En este caso `x1` e `y1` son las coordenadas del vértice superior izquierdo y `x2` e `y2` son las del vértice inferior derecho. Si quisiéramos dibujarlo con las esquinas redondeadas usaríamos la función `drawRoundRect()` que tiene los mismos parámetros que `drawRect()`. Si además de dibujar el perímetro se quiere que se dibuje el relleno, existen funciones tanto para rectángulos como para rectángulos con las esquinas redondeadas y son `fillRect()` y `fillRoundRect()` respectivamente, y ambos tienen también los mismos parámetros que `drawRect()`. De igual modo que cambiábamos de color el texto cuando escribíamos en pantalla, el color utilizado para realizar el relleno viene dado por el que se haya seleccionado mediante la llamada a la función `setColor()`.
En el caso de las circunferencias se dispone de la función:

```
drawCircle(x,y,radio);
```

donde como habrá supuesto el lector `x` e `y` indican el centro y el parámetro restante el radio de la circunferencia. También en este caso existe una función para dibujar la circunferencia con relleno `fillCircle()` y el color su relleno viene dado mediante la llamada previa a la función `setColor()`.
En el siguiente ejemplo se mostrará cómo utilizar estas funciones. Se dibujarán diez cuadrados concéntricos que irán aumentando de tamaño, luego se limpiará la pantalla y entonces se comenzará a dibujar cuadrados concéntricos pero rellenos y con las esquinas redondeadas; por último se limpiará

de nuevo la pantalla y se dibujarán de modo concéntrico circunferencias que irán aumentando su tamaño y se dibujarán con relleno y sin relleno de una forma intercalada.

```
#include "UTFT.h"

UTFT myTFT(ILI9325C,19,18,17,16); // para Uno

int xCenter;
int yCenter;

void setup(){
   myTFT.InitLCD(LANDSCAPE);
   myTFT.clrScr();
   randomSeed(analogRead(0));
   xCenter =  myTFT.getDisplayXSize() /2;
   yCenter =  myTFT.getDisplayYSize() /2;
}

void loop(){

   // rectángulos
   myTFT.clrScr();
   myTFT.setColor(255, 0, 156);
   for (int i=0; i<10; i++){
      myTFT.drawRect(xCenter - i * 10,yCenter  - i* 10,xCenter + i * _
            10,yCenter + i * 10);
      delay(500);
   }

   // rectángulos con relleno
   myTFT.clrScr();
   myTFT.setColor(10, 100, 156);
   for (int i=0; i<10; i++){
      myTFT.fillRoundRect (xCenter - i * 10,yCenter  - i* 10,xCenter + i * _
            10,yCenter + i * 10);
      delay(500);
   }

   // círculos
   myTFT.clrScr();
   // fondo gris
   myTFT.fillScr(VGA_GRAY);
   for (int r=5; r<100; r+=5){
      myTFT.setColor(VGA_AQUA);
      // circunferencia
      myTFT.drawCircle(xCenter,yCenter,r);
      delay(150);
      myTFT.setColor(VGA_RED);
      // circunferencia con relleno
      myTFT.fillCircle(xCenter,yCenter,r);
      delay(75);
   }
}
```

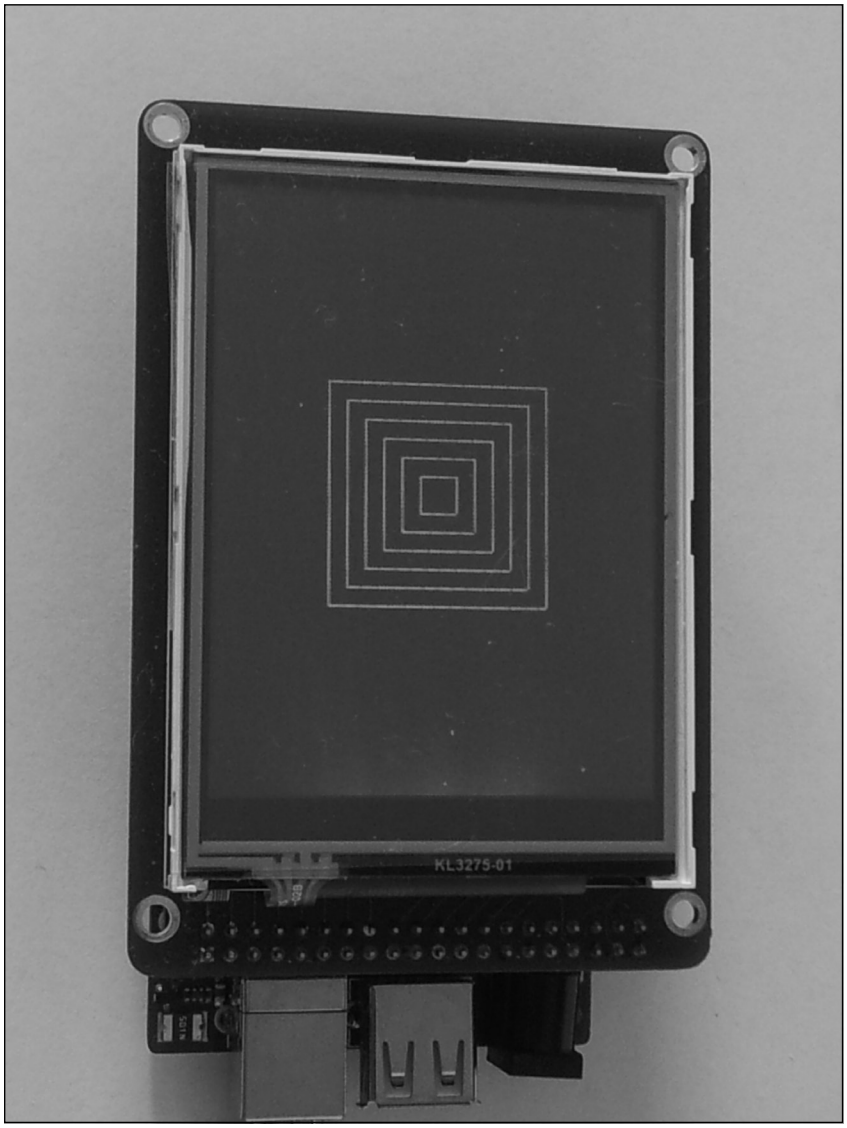


Figura 14.12. Figuras geométricas en TFT.

Nótese que además de haber seleccionado colores mediante la tripleta RGB, se ha seleccionado algún color mediante su constante, por ejemplo:

```
myTFT.fillScr(VGA_GRAY);
```

donde se le indica al microcontrolador que rellene todo el fondo de la pantalla con un color gris.

Todas las pantallas TFT, incluso las más económicas, tienen una resolución suficiente como para mostrar imágenes. La librería UTFT también proporciona funciones para poder trabajar con archivos gráficos pero previa utilización, deben tratarse para que sea viable su uso por parte del microprocesador y la librería.
Los archivos gráficos que se quieran mostrar por pantalla deben estar en formato RGB565, que se trata de un formato RGB de 16 bits sin canal alfa; para ello disponemos de una herramienta dentro del directorio `tools` de la librería UTFT o bien podemos acceder (entre otras) a la Web `http://www.henningkarlsen.com/electronics/t_imageconverter565.php` para realizar la conversión online.

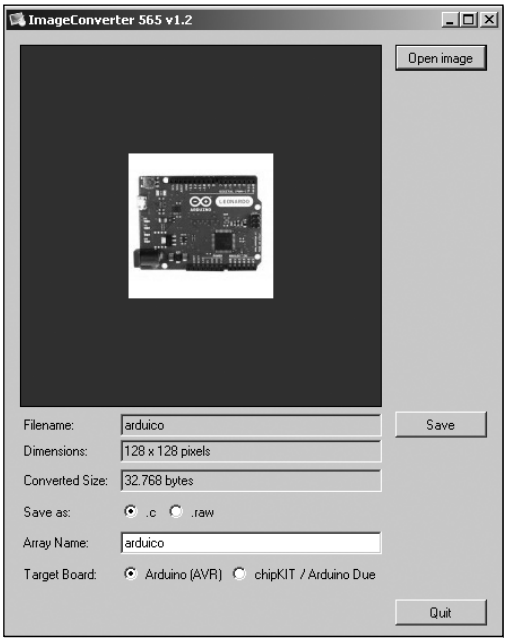


Figura 14.13. Programa de conversión a RGB565.

Al convertir la imagen podemos obtener el resultado en modo `.c` o en modo `.raw`. Para nosotros lo que mejor nos viene es el modo `.c`, donde obtendremos un array con los bytes necesarios para dibujar la imagen. Pulsando sobre el botón S**ave** se transformará la imagen en un array de bytes y se guardará en un fichero con extensión `.c`. Este archivo tiene la forma:

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

prog_uint16_t arduico[0x2667] PROGMEM ={
```

```
0xFFDF, 0xFFFF, 0xFFFF, 0xFFDE, 0xFFFF, 0xFFBD, 0xEF3B, 0xFFBB, 0xEF39,
0xFF7A, 0xFF3B, 0xE73A, 0xEEFA, 0xF73B, 0xEED9, 0xE739,   // 0x0010 (16)
0xEF1A, 0xE6F9, 0xE6D9, 0xEEF9, 0xEED9, 0xE6FA, 0xDF1C, 0xDF5E, 0xDEFE,
0xE6DC, 0xF75C, 0xD6FB, 0xD73B, 0xDF5B, 0xDF1B, 0xD71B,   // 0x0020 (32)
0xD6DB, 0xCED9, 0xDF5B, 0xD6DB, 0xCE9A, 0xCEB9, 0xCEFA, 0xCE7A, 0xCE9A,
0xC6B9, 0xCEDA, 0xD67B, 0xD67A, 0xC679, 0xC679, 0xD65A,   // 0x0030 (48)
0xC67A, 0xCE59, 0xCE59, 0xC659, 0xBDF8, 0xCE58, 0xCE38, 0xC639, 0xBE39,
0xC619, 0xBDF8, 0xC659, 0xC638, 0xC5F8, 0xC5F8, 0xBE18,   // 0x0040 (64)
...
```

En el podemos ver que se genera un array de tipo `prog_uint16_t` con el nombre proporcionado durante la creación y que se mantendrá guardado en el espacio de memoria Flash en lugar de hacerlo en SRAM con tal de ahorrar espacio de memoria de ejecución. Cada uno de las entradas del array son dos bytes conteniendo la información del color RGB del byte a mostrar en modo 5 bits R, 6 bits G, y 5 bits B. Una vez tenemos el gráfico que puede ser manejado por el *sketch* Arduino, pasemos a ver qué podemos hacer con él.
La librería UTFT ofrece una función para dibujar estas imágenes en pantalla y dependiendo de los parámetros informados se mostrarán realizando alguna transformación o no. La función y sus posibles parámetros son:

```
myTFT.drawBitmap(x,y,ancho,alto,datos,escala);
myTFT.drawBitmap(x,y, ancho,alto,datos,grados,centrox,centroy);
```

La primera llamada permite dibujar imágenes escalándolas; los parámetros son `x` e `y` para las coordenadas del vértice superior izquierdo, `ancho` y `alto` para indicar los pixeles que tiene la imagen original de la cual hemos obtenido el array, `datos` es el array conteniendo los datos a dibujar y `escala` es un parámetro optativo que indica el factor de escala, cada pixel será dibujado con un factor como indique la escala. En el caso de no querer escalar la imagen podemos no indicar el parámetro `escala`. La segunda llamada por contra, permite dibujar la imagen con rotación, los parámetros comunes son lo mismo que en la primera llamada, `grados` indica el número de grados a girar la imagen (este parámetro debe tomar valores entre 0 y 359, es decir no se da en radianes sino en grados sexagesimales) y `centrox` y `centroy` especifican las coordenadas del punto que se tomará como centro para realizar la rotación (estas coordenadas van referidas a las coordenadas del punto correspondiente a la esquina superior izquierda de la imagen). Tanto `grados` como `centrox` y `centroy` son parámetros optativos.
Para ver el funcionamiento de estas funciones realizaremos un *sketch* en el que se cargará una imagen previamente convertida con la herramienta que acompaña la librería y se mostrará primero tal cual es en realidad, después se mostrará girada en 15°, 30° y 45° y por último la acabaremos escalando a varios tamaños.

```
#include "UTFT.h"
#include "avr/pgmspace.h"

UTFT myTFT(ITDB32S, 38,39,40,41 );  // para Mega

// imagen en RGB565
// se debe cambiar por la generada
prog_uint16_t arduico[0xC40] PROGMEM ={
0xFFFF, 0xFFFF, 0xF79D, 0xDE78, 0xCDF5, 0xDE36, 0xB5B4, 0xBD74, 0xB573,
0xBDB4, 0xB553, 0xBD32, 0xAD75, 0xADB8, 0xB576, 0xADB6,   // 0x0010 (16)
0xADB5, 0xA595, 0xA575,
.....
};

void setup(){
   myTFT.InitLCD(LANDSCAPE);
   myTFT.clrScr();
}

void loop(){
   myTFT.clrScr();
   // imagen tal cual
   myTFT.drawBitmap(20,20,64,49, arduico);
   delay(2000);

   // imagen rotada
   for (byte i = 0; i < 3; i++){
      myTFT.clrScr();
      myTFT.drawBitmap(20,20,64,49, arduico, (i+1)*15, 0,0);
      delay(2000);
   }

   // escala 1:2
   myTFT.drawBitmap(20,20,64,49, arduico,2);
   delay(2000);
   myTFT.clrScr();
   // escala 1:3
   myTFT.drawBitmap(20,20,64,49, arduico,3);
   delay(2000);
   myTFT.clrScr();
   // escala 1:4
   myTFT.drawBitmap(20,20,64,49, arduico,4);
   delay(2000);
   myTFT.clrScr();
   //  escala 1:5
   myTFT.drawBitmap(20,20,64,49, arduico,5);
   delay(5000);
}
```

Cuando realicemos este *sketch*, hay que tener en cuenta que el array de datos de la imagen en este ejemplo se llama `arduico` pero que en cada caso será distinto, dependiendo de lo informado a la hora de crear el array mediante la herramienta; esto implica que se deben cambiar todas la ocurrencias de `arduico` en el *sketch* por el nombre del nuevo array.

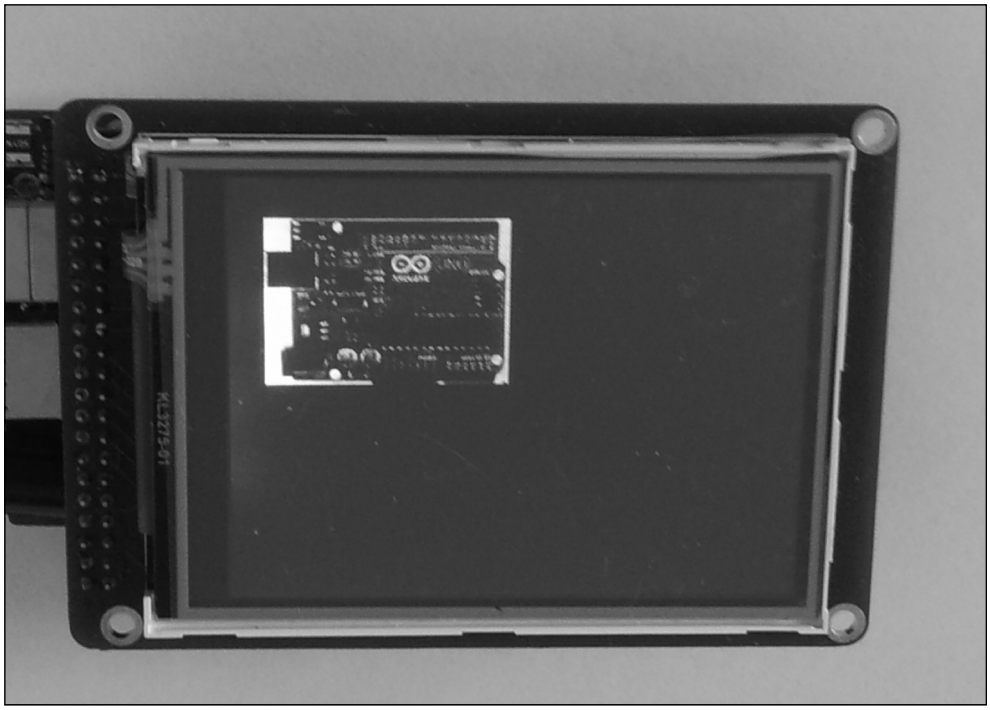


Figura 14.14. Gráficos en TFT.

Pantallas TFT Táctiles

Con el abaratamiento de la fabricación de pantallas TFT, actualmente la mayor parte de los fabricantes las ofrecen que además son capaces de reaccionar a pulsaciones, son las llamadas pantallas táctiles y que podemos encontrarlas de dos tipos: resistivas y capacitivas.

Las pantallas resistivas suelen ser más económicas y de peor resolución táctil que las capacitivas, y normalmente suelen venir acompañadas de un puntero a modo de lápiz de plástico con el cual pulsar, aunque también es posible pulsarlas con el dedo.

Las pantallas resistivas se componen de varias capas, entre ellas las más importantes son dos capas metálicas muy finas conductiva y resistiva que se encuentran separadas por muy poco espacio. Cuando alguien toca la pantalla, las capas se conectan en ese punto y se genera algo semejante a dos divisores de tensión (uno por cada eje) como los que se han visto a lo largo del libro, y esto genera una variación en la corriente eléctrica que es tomada como una pulsación y se envía al controlador para que calcule el punto exacto.

Por su parte las pantallas capacitivas se componen de un aislante como el cristal, recubierto por una fina capa de material conductor transparente que suele ser una mezcla de óxido de estaño y óxido de indio.
Puesto que el cuerpo humano es conductor, al tocar la pantalla se produce una distorsión del campo electrostático del cuerpo, que puede medirse dado que cambia su capacitancia. A partir de esta variación se pueden utilizar diferentes tecnologías para localizar el punto de contacto y enviárselas al microcontrolador para su proceso.
Los punteros de las pantallas resistivas son de plástico y no conducen la electricidad, por eso no funcionan cuando los usamos con una pantalla capacitiva y hay que comprar punteros con espumas o gomas especiales. Si tiene pensado utilizar un clavo o algo semejante como puntero por ser conductor, es muy probable que tampoco le funcione ya que la mayoría de los controladores de este tipo de pantallas determinan que se ha producido una pulsación cuando se detecta que objeto que pulsa es algo semejante a un dedo y normalmente viene dado por la detección de una pulsación por un objeto que tiene un diámetro de unos 4 mm... es decir la superficie de contacto de un dedo al pulsar.
Para poder detectar las pulsaciones en las pantallas táctiles con Arduino podemos optar por hacer toda la programación nosotros u optar una vez más por el uso de unas librerías; como es de esperar, recomiendo la segunda opción.
Dentro de las librerías disponibles existe una del creador de la utilizada anteriormente llamada UTouch y que se puede descargar de forma gratuita desde `http://henningkarlsen.com/electronics/library.php?id=55`.
Esta librería solamente tiene funcionalidad relativa a las pulsaciones, es decir para poder trabajar con las pantallas seguimos necesitando tener la librería UTFT. Nuevamente no todas las pantallas están soportadas pero si una gran mayoría. Igual que el resto de las librerías, una vez descargadas, debemos descomprimirlas y copiarlas al directorio `libraries` dentro del directorio de instalación del entorno de programación Arduino.
Para poder utilizar esta librería en el *sketch* se debe incluir mediante la línea de código:

```
#include <UTouch.h>
```

De la misma forma que con UTFT, debemos crear una instancia de la clase `UTouch` que será el nexo de unión con la pantalla y sobre la variable que trabajemos.
La creación de la instancia se realiza mediante la llamada:

```
UTouch nombre_variable(pinClock, pinCS, pinIn, pinOut, pinIRQ);
```

En este caso `pinClock` es el pin para los pulsos de reloj, `pinCS` es para la selección de chip, `pinIn` y `pinOut` para entrada y salida respectivamente y por último `pinIRQ` para el manejo de interrupciones.
Los pines utilizados varían dependiendo de la pantalla y de si estamos trabajando con una tarjeta Arduino Uno o bien una Arduino Mega, pero normalmente las creaciones de las instancias, dependiendo de la tarjeta Arduino utilizada serían:

```
UTouch      myTouch(6,5,4,3,2); // para Arduino MEGA
UTouch      myTouch(15,10,14,9,8); // para Arduino Uno
```

La inicialización de la instancia se realiza dentro de la función `setup()`.

```
myTouch.InitTouch();
myTouch.setPrecision(PREC_MEDIUM);
```

Al igual que con UTFT, la función `InitTouch()` puede tener como parámetro la constante `PORTRAIT` o `LANDSCAPE` para indicar que se quiere trabajar con la pantalla en modo vertical u horizontal respectivamente, en caso de no informar nada, el valor por defecto es `LANDSCAPE`.
En el caso de la función `setPrecision()` configura la precisión con la que se leerá la posición de la pulsación; los valores posibles de mayor a menor precisión son: `PREC_EXTREME`, `PREC_HI`, `PREC_MEDIUM` y `PREC_LOW`. Cuanto mayor sea la precisión, más tardará en leerse el dato y por lo tanto se debe tener cuidado sobre todo si se usa `PREC_EXTREME` cuando se van a realizar varias pulsaciones seguidas ya que podrían perderse datos.
Las lecturas de las pulsaciones de pantalla que se van realizando por parte de la librería se irán almacenando en un buffer; para consultar si existen datos en él, se utiliza la función:

```
myTouch.dataAvailable()
```

que devolverá `true` si existen datos para leer. En caso de que existan datos accederemos a ellos mediante la llamada:

```
myTouch.read();
```

Cuando se han leído los datos mediante la función `read()`, entonces en la instancia de la clase `UTouch` se tienen accesibles las coordenadas mediante sendas llamadas a:

```
x = myTouch.getX();
y = myTouch.getY();
```

Para comprobar el funcionamiento de las pulsaciones, realizaremos un *sketch* donde se irá leyendo la pulsación de la pantalla y se pintará el pixel sobre el que se ha pulsado, de manera que sería como si fuéramos pintando en la pantalla.

Para dibujar los puntos volveremos a utilizar una función de la librería UTFT, en concreto:

```
myTFT.drawPixel(x, y);
```

Nota:

La librería UTouch solamente tiene funcionalidad para detectar pulsaciones, para mostrar textos e imágenes por pantalla sigue siendo necesaria la librería UTFT.

El listado del *sketch* para pintar las pulsaciones quedaría:

```
#include <UTFT.h>
#include <UTouch.h>

// Para Arduino Uno
// UTFT        myGLCD(ILI9325C,19,18,17,16);    // cambiar el modelo según
                                                // necesidades
// UTouch      myTouch(15,10,14,9,8);

// Para Arduino Mega
UTFT        myTFT(ITDB32S, 38,39,40,41);        // cambiar el modelo según
                                                // necesidades
UTouch      myTouch(6,5,4,3,2);

void setup(){
   // iniciación de la pantalla
   myTFT.InitLCD();
   myTFT.clrScr();

   // iniciación de las pulsaciones
   myTouch.InitTouch();
   myTouch.setPrecision(PREC_MEDIUM);

   // color del pixel rojo
   myTFT.setColor(255, 0, 0);
}

void loop(){
   long x;
   long y;

   while (myTouch.dataAvailable()){
      myTouch.read(); //lectura de datos
      x = myTouch.getX();
      y = myTouch.getY();
      if ((x!=-1) && (y!=-1)){
         myTFT.drawPixel(x, y);
      }
   }
}
```

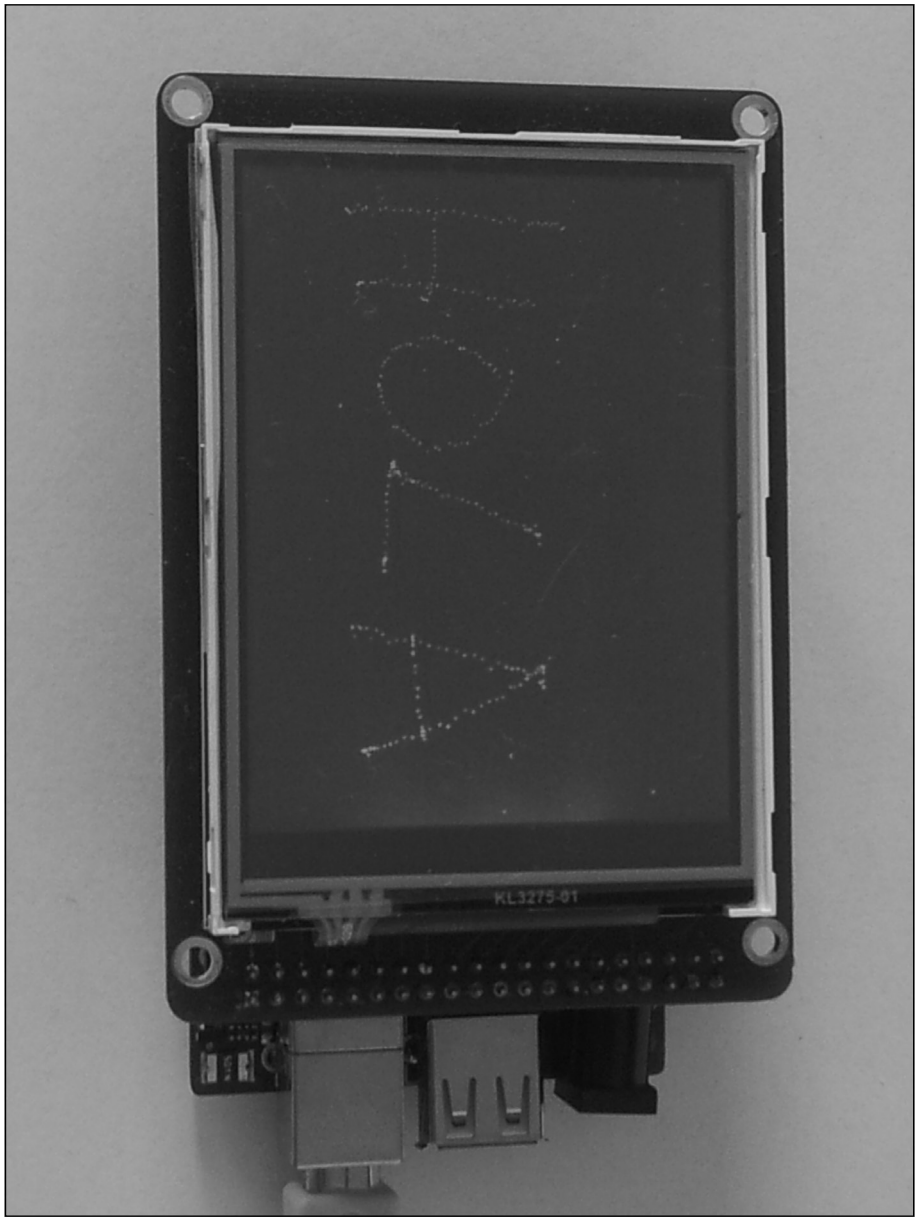


Figura 14.15. Trazo de puntos en TFT.

Antes de ejecutar el *sketch* no hay que olvidarse ajustar el modelo y también los pines de inicialización tanto de la tarjeta Arduino como de la pantalla utilizada.

El trazo obtenido de este modo es a modo de camino de miguitas de pan, no parece que estemos dibujando. Para mejorar el aspecto del trazo crearemos un *sketch* donde se dibujen líneas en lugar de puntos. La función para dibujar líneas corresponde a la librería UTFT:

```
myTFT.drawPixel (x1, y1, x2, y2);
```

En este caso las coordenadas que se necesitan son las del punto de origen de la línea y las del punto final.
En el *sketch* lo que se hará será recoger la pulsación y dibujar una línea desde el punto leído por la pulsación hasta el punto leído en la pulsación anterior; en caso de ser la primera pulsación, simplemente se debe guardar como última pulsación recibida pero no se debe pintar nada.

```
#include <UTFT.h>
#include <UTouch.h>

// Para Arduino Uno
// UTFT        myGLCD(ILI9325C,19,18,17,16);    // cambiar el modelo según
                                                // necesidades
// UTouch      myTouch(15,10,14,9,8);

// Para Arduino Mega
UTFT        myTFT(ITDB32S, 38,39,40,41);        // cambiar el modelo según
                                                // necesidades
UTouch      myTouch(6,5,4,3,2);

// última pulsación
long lastX = -1;
long lastY = -1;

void setup(){
   // iniciación de la pantalla
   myTFT.InitLCD();
   myTFT.clrScr();

   // iniciación de las pulsaciones
   myTouch.InitTouch();
   myTouch.setPrecision(PREC_MEDIUM);

   // color del pixel rojo
   myTFT.setColor(255, 0, 0);
}

void loop(){
   long x;
   long y;

   while (myTouch.dataAvailable()){
      myTouch.read(); // lectura de datos
      x = myTouch.getX();
      y = myTouch.getY();
```

```
      if (((x!=-1) && (y!=-1)) && (lastX != -1 && lastY != -1) ){
         myTFT.drawLine (x, y, lastX, lastY);
      }
      // guardar última pulsación
      lastX = x;
      lastY = y;

   }
}
```

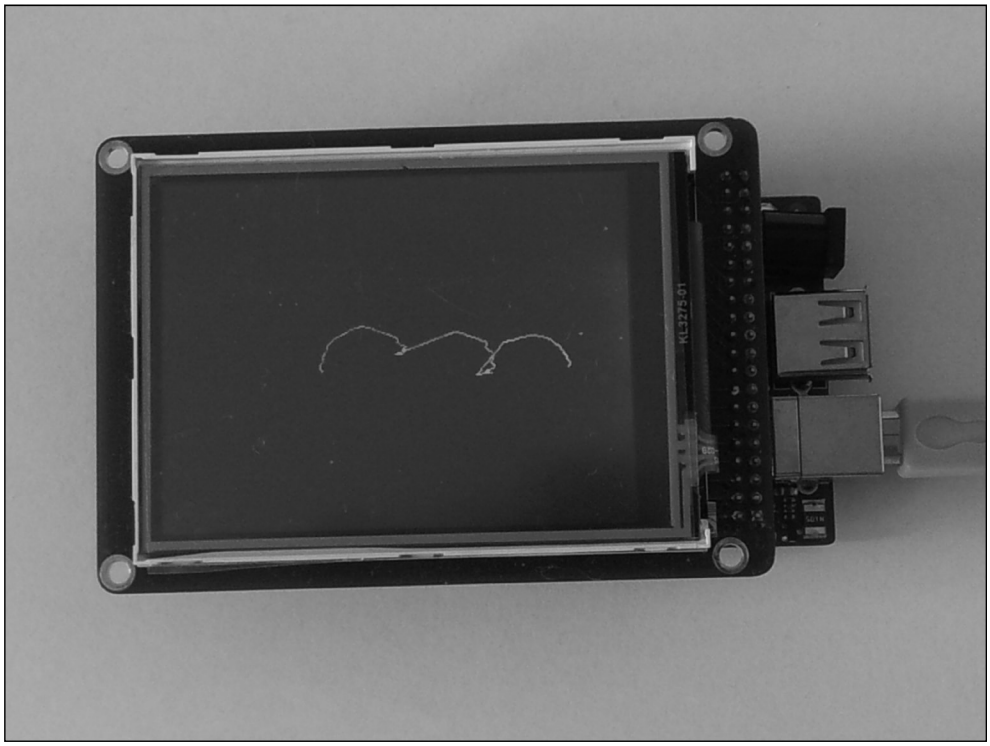


Figura 14.16. Trazo de líneas en TFT.





A

Resistencias

Se conoce como resistencia eléctrica a la oposición que ofrecen los materiales a que los electrones se desplacen a través de ellos. La unidad de resistencia en el sistema internacional es el Ohmio o simplemente Ohm y se representa por la letra griega omega Ω. Esta unidad se denomina así en honor al físico alemán George Ohm descubridor del principio que lleva su nombre. La ley de Ohm se define como que la resistencia de un material es la razón entre la diferencia de potencial y la corriente eléctrica que atraviesa dicho material y puede expresarse como:

$$R = \frac{V}{I}$$

Cada material posee intrínsecamente una resistividad o coeficiente de proporcionalidad que define la resistencia que tendrá el material representado por la letra rho ρ, junto con su longitud y grosor. La resistencia total de un material depende directamente de su resistividad y su longitud e inversamente de la sección del mismo, o dicho en otras palabras, cuanto más largo sea el material mayor resistencia tendrá; por otro lado cuanto más grueso sea menor resistencia ofrecerá.

$$R = \rho \frac{\ell}{S}$$

La unidad inversa al Ohmio es el Siemens y se denota por la letra S en honor al físico alemán Werner von Siemens y mide la conductancia de un elemento.
Las resistencias eléctricas son muy utilizadas en montajes electrónicos para ajustar los niveles de tensión y corriente que pasan por los distintos elementos que componen el circuito.
Normalmente en los circuitos electrónicos nos encontramos las resistencias asociadas entre sí.
Aunque son muchas las distintas asociaciones de resistencias que se pueden encontrar (en triángulo, en estrella, en H...) a continuación veremos las dos asociaciones más comunes: en serie y en paralelo.

Resistencias en serie

Las resistencias se encuentran asociadas en serie cuando al aplicar al conjunto una diferencia de potencial, la corriente que recorre todas las resistencias es la misma.

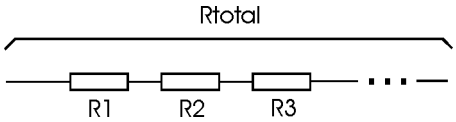


Figura A.1. Resistencias en serie.

Según la ley de Ohm, en este caso se tendría la ecuación:

$$V = R_1 * I + R_2 * I + R_3 * I + R_4 * I = I * (R_1 + R_2 + R_3 + R_4) = I * R_e$$

Por lo que podríamos sustituir todas las resistencias de la rama serie por una resistencia equivalente cuyo valor sería es la suma de los valores de todas las resistencias de la rama serie.

Resistencias en paralelo

Las resistencias se encuentran asociadas en paralelo cuando tienen sus dos terminales comunes entre ellas de manera que cuando se le aplica una diferencia de potencial en ellos, todas las resistencias presentan la misma caída de tensión.

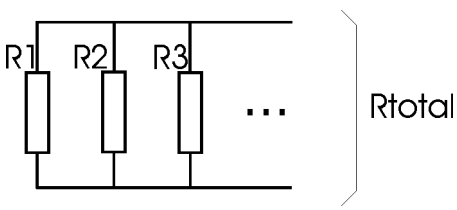


Figura A.2. Resistencias en paralelo.

Según la primera ley de Kirchhoff que también es conocida como ley de las corrientes: en cualquier nodo de un circuito eléctrico, la suma de las corrientes que entran en dicho nodo es igual a la suma de las corrientes que salen, o dicho de otra forma: la suma de todas las corrientes que pasan por el nodo es igual a cero.
Luego la corriente que entra en el nodo será igual a la suma de las corrientes que pasan por cada una de las resistencias del conjunto.

$$I = I_1 + I_2 + I_3 + I_4 = \frac{V}{R_1} + \frac{V}{R_2} + \frac{V}{R_3} + \frac{V}{R_4} = V(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \frac{1}{R_4})$$

Si se dispone de una resistencia equivalente, generaría la misma intensidad para esa diferencia de potencial aplicado:

$$I = \frac{V}{R_e}$$

Entonces igualando ambas ecuaciones se tendría que la resistencia equivalente en este caso sería:

$$\frac{1}{R_e} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \frac{1}{R_4}$$

Así pues la resistencia equivalente de una asociación en paralelo es igual a la inversa de la suma de las inversas de las resistencias de cada una de las ramas, o dicho de otro modo, la conductancia equivalente de unas resistencias en paralelo es igual a la suma de las conductancias de las resistencias de cada rama.

Codificación de valores de las resistencias

A la hora de realizar los montajes electrónicos, deberemos ser capaces de conocer el valor nominal de cada una de las resistencias que se van a utilizar. Una manera sería por medio de un multímetro en posición de Óhmetro e ir midiendo, pero esto no sería eficiente. Dependiendo del tipo de resistencia, existen unos estándares para marcar las resistencias con su valor nominal y poder conocerlo a simple vista.

Resistencias SMT

Si las resistencias son de montaje superficial son las llamadas resistencias SMT (*Surface Mount Technology*, tecnología de montaje en superficie) o SMD y están marcadas con tres caracteres de los cuales uno puede ser una letra; para marcarlas se utilizan tolerancias estándar. Son las resistencias que podemos ver en las placas de circuitos integrados.

- Si los tres caracteres son dígitos, los primeros dos dígitos representan los primeros dos dígitos significativos y el tercer dígito representa una potencia de diez (el número de ceros). Por ejemplo si la marca fuera 332, los ohmios serían 3200.

- Si el segundo carácter es una R, el primero indica la primera cifra del valor nominal, la R el punto decimal y la tercera la primera cifra decimal. Por ejemplo la serigrafía 2R4 serían 2.4 Ω.
- Si el primer carácter es R, significa que los dos siguientes son decimales. Por ejemplo si está marcada con R15 es que la resistencia es de 0.15 Ω.

Figura A.3. Resistencias de montaje superficial.

Hay resistencias de montaje en superficie en las que aparecen 4 caracteres en vez de tres y todos serán dígitos; en este caso se trata de resistencias de precisión y el valor nominal de la resistencia viene dado por los tres primeros dígitos que son números significativos de la resistencia y el cuarto es el número de ceros a añadir (o dicho de otro modo a que potencia se debe elevar 10 y multiplicar los números significativos por el resultado). Por ejemplo 4462 serían 44600Ω o 44.6KΩ.

Resistencias through-hole

Las resistencias *through-hole* (a través de agujero), de empaquetado axial o simplemente axiales son las típicas resistencias con unas bandas de colores, como las que se utilizan en el libro para realizar prototipos. Se denominan *through-hole* porque para su montaje es necesario atravesar un agujero. Para diferenciar el valor nominal de este tipo de resistencias se utilizan una serie de bandas de colores con distinto significado según la posición, siguiendo un código internacional que asigna un valor numérico a cada color. Además existen bandas que corresponden al multiplicador, tolerancia y coeficiente de temperatura pero no tienen porqué estar todas informadas. Existen resistencias tanto de 4, como de 5 y 6 bandas. Lo más normal es encontrar las resistencias de 4 bandas y las de 5 bandas que suelen ser de precisión; las de 6 bandas son menos comunes. En las resistencias de 4 bandas, éstas corresponden a:

1. Decenas.
2. Unidades.
3. Multiplicador.
4. Tolerancia (normalmente dorado y plateado solamente).

En las resistencias de 5 bandas:

1. Centenas.
2. Decenas.
3. Unidades.
4. Multiplicador.
5. Tolerancia.

En las de 6 bandas todas las columnas están presentes.
Los valores correspondientes podemos verlos en la siguiente tabla.

Tabla A.1. Código de colores de resistencias.

Color	Centenas	Decenas	Unidades	Multiplicador	Tolerancia	Coef. de temp.
Negro	0	0	0	1	-	-
Marrón	1	1	1	10	±1%	100ppm/°C
Rojo	2	2	2	100	±2%	50ppm/°C
Naranja	3	3	3	1 000	-	15ppm/°C
Amarillo	4	4	4	10 000	±4%	25ppm/°C
Verde	5	5	5	100 000	±0,5%	20ppm/°C
Azul	6	6	6	1 000 000	±0,25%	10ppm/°C
Morado	7	7	7	10000000	±0,1%	5ppm/°C
Gris	8	8	8	100000000	±0.05%	1ppm/°C
Blanco	9	9	9	1000000000	-	-
Dorado	-	-	-	0,1	±5%	-
Plateado	-	-	-	0,01	±10%	-
Ninguno	-	-	-	-	±20%	-

A continuación se muestra una tabla con ejemplos:

1ª banda	2ª banda	3ª banda	4ª banda	5ª banda	6ª banda	Valor
Rojo	Amarillo	Verde	Rojo	-	-	2M4 ± 2%
Morado	Amarillo	Rojo	Marrón	-	-	7k4 ± 1%

1ª banda	2ª banda	3ª banda	4ª banda	5ª banda	6ª banda	Valor
Rojo	Rojo	Marrón	Verde	-	-	220R ± 0.5%
Azul	Marrón	Naranja	Rojo	Verde	-	61k3 ± 0.5%
Rojo	Rojo	Rojo	Negro	Morado	-	222R ± 0.10%
Amarillo	Gris	Rojo	Marrón	Oro	Violeta	4k82 ± 5% 5ppm/°C
Marrón	Verde	Azul	Negro	Plata	Amarillo	156R ± 10% 25ppm/°C

Figura A.4. Resistencias axiales.

Resistencias SIL

Las resistencias SIL (*Single in line*, solo en línea) son unas resistencias semejantes a las *through-hole* salvo que vienen varias empaquetadas en el mismo elemento.

Cada uno de los componentes tiene un terminal que es común a todas las resistencias y el resto de terminales permiten conectar cada una de las resistencias con el común. Durante su utilización no es necesario que se conecten todos los terminales, los no conectados implemente no se usan pero no afectan al funcionamiento.

El terminal común en estos elementos suele diferenciarse con una marca específica, normalmente un punto o bien un triángulo sobre él. Los valores nominales suelen venir dados por tres dígitos, siendo los dos primeros números significativos del valor nominal y el tercero el número de ceros a añadir o la potencia de 10 por la que multiplicar.

Aunque es posible encontrar resistencias SIL no normalizadas, los valores disponibles están reglados: 22, 27, 33, 39, 47, 56, 68, 82, 100, 120, 150, 220, 270, 330, 390, 470, 560, 680, 820, 1K, 1.2K, 1.5K, 1.8K, 2K, 2.2K, 2.7K, 3.3K, 3.9K, 4.7K, 5.6K, 6.8K, 8.2K, 10K, 12K, 15K, 18K, 20K, 22K, 27K, 33K, 39K, 47K, 56K, 68K, 82K, 100K, 120K, 150K, 180K, 220K, 270K, 330K, 390K, 470K, 560K, 680K, 820K, 1M.

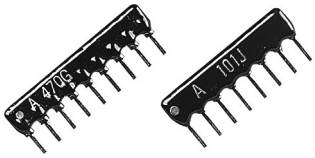


Figura A.5. Resistencias SIL.

Por ejemplo un SIL con la marca 470 serían 47 Ω y uno con la 101 serían 100 Ω.

B

Sistemas numéricos

Cuando contamos desde 0, incrementando una unidad cada vez, al llegar a las 9 unidades, se han agotado los símbolos disponibles, y si queremos seguir contando no disponemos de un nuevo símbolo para representar la cantidad que hemos contado y debemos entonces utilizar dos símbolos para representar este nuevo número; así es como nos enseñaron a contar. Normalmente estamos acostumbrados a trabajar con números en base 10, es decir con números del 0 al 9, pero en electrónica es muy común trabajar con otras bases, concretamente en las bases 2 y 16 conocidas como sistema binario y sistema hexadecimal respectivamente. Vamos a hacer una pequeña introducción a estas bases de numeración y entraremos un poco más en profundidad en las transformaciones y operaciones en base 2 por ser la más usada y así familiarizarnos con ellas.

Sistema binario

En el sistema binario tan sólo se utilizan dos números para representar cualquier cantidad, el 0 y el 1. Los números en este sistema pueden almacenarse de muchas formas (siempre que utilicemos sólo 0 y 1) y es lo que se llaman codificaciones, más adelante veremos algunas de ellas como la BCD Aiken.
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 bien 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.
El mayor número de cantidades que puede representar en binario un grupo de bits viene dado por el número de bits que haya, en concreto 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 bit adicional.

1 bit	2 bits	3 bits	4 bits
0	00	000	0000
1	01	001	0001

1 bit	2 bits	3 bits	4 bits
	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 lo tanto de elementos a representar con n bits son 2^n.

Cada posición del bit tiene un peso dentro del número final representado; normalmente se toma que cuanto más a la derecha estén menor peso tienen, de modo semejante a como manejamos los números en base decimal; en el número 23 el 2 tiene más peso (representa al 20, las decenas) que el 3 (representa las unidades). Al bit de mayor peso se le conoce como MSB (*Most Significant Bit*, Bit de Mayor Significado) y al de menor peso LSB (*Least Significant Bit*, Bit de Menor Significado). Para conocer la cantidad representada por un número (sea la base que sea) se puede aplicar la fórmula:

$$N = A_n*B^n + A_{(n-1)}*B^{(n-1)} + A_{(n-2)}*B^{(n-2)} + \ldots + A_2*B^2 + A_1*B^1 + A_0*B^0$$

Siendo A los dígitos de representación, B la base del sistema numérico, n el número de dígitos menos 1 y N la cantidad decimal. Así por ejemplo 11101 sería:

$$N = 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 29$$

Si hubiera números decimales se procedería de igual manera, teniendo en cuenta que el exponente sería negativo. Por ejemplo 11101.101:

$$N = 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = 29.635$$

La conversión de binario a decimal es muy sencilla, simplemente se deben sumar los pesos asociados a los bits y multiplicarlos por el valor de los bits; por ejemplo si se quisiera saber qué cantidad decimal representa el número 10010101.1, valdría con sumar los pesos de las posiciones que tengan un 1, para facilitar la tarea crearemos una tabla donde escribimos los pesos de cada bit (recordemos que el valor de los pesos viene dado por 2^n siendo n la posición que ocupan comenzando desde la derecha y la primera posición no decimal es la 0).

Peso	128	64	32	16	8	4	2	1	0.5
	1	0	0	1	0	1	0	1	1

Sumamos todos los pesos que tengan el bit a 1 para obtener la representación decimal:

```
N = 128 + 16 + 4 + 1 + 0,5 = 149,5
```

De manera inversa, para obtener el número binario lo que se hace es dividir el número decimal entre la base que en este caso es 2 y nos quedamos con el resto o hasta que no se pueda dividir más, para ir componiendo el número binario de bit menos significativo a más significativo. Veamos un ejemplo semejante al anterior.
Para transformar 149.685 en 10010101.1011:

```
149 / 2 = 74 resto 1
74  / 2 = 37 resto 0
37  / 2 = 18 resto 1
18  / 2 = 9  resto 0
9   / 2 = 4  resto 1
4   / 2 = 2  resto 0
2   / 2 = 1  resto 0
1
```

Para la parte decimal multiplicaremos por dos y para formar el número nos quedamos entonces con la parte entera que resulte, en nuestro caso será lo siguiente:

```
0.6875 * 2 = 1.375
0.375  * 2 = 0.75
0.75   * 2 = 1.5
0.5    * 2 = 1
```

Generamos el número tomando los restos de "abajo a arriba" para formar la parte entera y la parte decimal la formamos con los enteros leídos de "arriba a abajo"; así obtenemos el número 10010101.1011.

Operaciones binarias

Las operaciones que se pueden realizar con números binarios son las mismas que con los números en base decimal, aunque al no estás acostumbrados a trabajar con esta base puede que nos resulte más difícil. Veamos las operaciones aritméticas más comunes.

Suma

Para sumar números binarios se debe tener en cuenta la tabla:

```
0+0=0
0+1=1
1+0=1
1+1=0 acarreo = 1
```

El acarreo es el "me llevo uno" que usábamos en el colegio, es como cuando se suma 4+9, que es 3 y me llevo 1 que irá a la siguiente decena. Para realizar la suma se procede del mismo modo que en decimal, de derecha a izquierda (de menos significativo a más significativo) teniendo en cuenta los acarreos. Por ejemplo:

```
10111 + 10010  = 101001
```

Resta

En cuanto a la resta funciona de la misma forma que en la resta decimal; de bit menos significativo a más significativo iremos restando bit a bit usando la tabla:

```
0-0=0
0-1=1 y debo 1
1-0=0
1-1=0
```

Del mismo modo que la suma comporta acarreo, la resta binaria tiene el "debo uno" como sucede en la resta decimal.
Por ejemplo:

```
10111 - 10010  = 000101
```

Multiplicación

La multiplicación funciona del mismo modo a cuando se aplica a los bits un "y" lógico, la tabla de uso es:

```
0 * 0 = 0
0 * 1 = 0
1 * 0 = 0
1 * 1 = 1
```

Si realizáramos la operación de multiplicación en decimal de 6 * 13 obtendríamos 78 y en binario debería ser igual 1101 * 110 = 1001110 vamos a ver cómo lo haríamos:

```
  1101
X 110
------
  0000
 1101
1101
-------
1001110
```

Una manera muy rápida de realizar multiplicaciones por 2 es desplazar los bits a la izquierda tantas posiciones como veces se quiera multiplicar por dos, por ejemplo si quisiéramos realizar la multiplicación 9 * 4 = 36 sería desplazar 9 dos posiciones a la izquierda dado que 4 es 2^2; entonces si a 9 = 1001 si lo desplazamos dos posiciones sería 100100 que en base decimal es 36.

División

La división es semejante al sistema decimal, comenzar a realizar grupos del dividendo de izquierda a derecha de modo que sean divisibles entre el divisor y calcular su resta, una vez hallada la resta ir "bajando" números del numerador y seguir dividiendo; una combinación de multiplicaciones y restas. Por ejemplo tomando los números anteriores 78 / 6 = 13 en binario sería:

```
1001110 | 110
-110    |------
------  | 1101
  111
 -110
 ----
   110
  -110
  ----
   000
```

De la misma forma que si se desplazaban los bits a la izquierda se obtenían multiplicaciones por 2, si se desplazan hacia la derecha lo que se obtienen son divisiones entre 2, y en caso de que la división no sea exacta se obtiene el resultado truncado, por ejemplo 9 /2 = 4.5, si desplazamos 9 una posición a la derecha obtendríamos 4: 9 en binario es 1001 desplazado es 100 que es 4.

Números negativos

Para guardar números negativos en binario existen múltiples formas; una de ellas es usar el bit de mayor peso (el MSB) como signo del número, utilizando 0 para números positivos y 1 para negativos; el resto de los bits, llamados magnitud, nos sirven para guardar el número en sí; tendremos n-1 bits para representar el número y 1 bit para el signo. Supongamos que utilizamos un byte, si sólo se utiliza el último bit para diferenciar los positivos de los negativos nos encontraríamos con un problema y es que 10000000 y 00000000 en binario serían -0 y 0, es decir tenemos dos representaciones para la misma cantidad, con lo que estamos perdiendo capacidad de almacenar números. Para 8 bits tendríamos que para la magnitud hay disponibles 7 bits, después podríamos representar 2^7, de -128 a 128. Aunque hay sistemas que utilizan este sistema de representación para trabajar, lo normal es utilizar uno que aproveche todas las combinaciones, entre ellos el más difundido es el denominado complemento a dos.

Complemento a dos

Mediante el complemento a dos, se pueden almacenar tanto números positivos como negativos aprovechando todas las combinaciones para representar números diferentes. Lo que hace esta representación es utilizar el primer bit de la izquierda como bit de signo, pero el número negativo se calcula en tres pasos una vez que lo tenemos en base binaria:

1. Complementamos el número cambiando los 0 por 1 y los 1 por 0.
2. Sumamos 1 al resultado.
3. Se añade el bit de signo.

Por ejemplo el número -35 sería:

1. El número 35 en binario en 7 bits: 0100011.
2. Complementado: 1011100.
3. Sumamos 1: 1011100 + 1 = 1011101.
4. Le añadimos el bit de signo 11011101.

Usando este sistema 00000000 representa al 0 y 10000000 representa a –128, esto lo conseguimos restando 1 a 10000000 = 01111111 y complementando obtendremos10000000 que es 128 (o complementando primero y sumando uno después), con lo que el rango es de -128 a 127 obteniendo una representación más que en binario natural con signo.

Cuando se está en complemento a dos, si se ha de extender el tamaño del dato (por ejemplo pasar de un *nibble* a un *byte*) se debe rellenar con 0 si son positivos y con 1 si son negativos en complemento a dos.

Número decimal	Complemento a dos	Complemento a dos en byte
3	0011	00000101
-3	1101	11111101

Aunque sea un poco lioso el complemento a dos, ofrece grandes ventajas a la hora de trabajar con operaciones de resta, ya que obtendremos el resultado directamente en complemento a dos. La única precaución que se ha de tener es que el acarreo generado por el bit de signo no se debe tener en cuenta.

```
9-3 = 6   01001 + 01101 = 00110
3-9 =-6   00101 + 10111 = 11010
```

Sistemas de representación

Cuando se representan números en una base, normalmente se asignan pesos a cada posición de la representación numérica para obtener la cantidad representada, pero no siempre es así. En binario existen múltiples formas de representar cantidades numéricas; además del sistema de representación binario natural que es el que se ha estado utilizando hasta ahora existen otros sistemas de representación que utilizan distintos pesos de bits u otras referencias para formar el número binario. Aunque existen muchos, nosotros veremos alguno de los más importantes: BCD natural, BCD exceso a 3, BCD Aiken y Gray.

BCD Natural

La codificación BCD (*Binary Coded Decimal*, codificación binaria decimal) es un sistema por el cual cada uno de los dígitos del número se representa por separado, de modo que cada 4 bits representarán un dígito. Este tipo de codificación es muy útil en sistemas en los que se tengan que mostrar los dígitos en pantalla como en displays de 7 segmentos ya que permiten trabajar dígito a dígito por separado sin necesidad de hacer operaciones previas.
En el caso de la codificación BCD natural, codificaremos cada número por separado usando binario natural en cada uno de ellos. En un *nibble* tendríamos $2^4 = 16$ números distintos, por lo que si solamente representamos del 0 al 9, tenemos una pérdida de 6 combinaciones por no ser utilizadas.

Decimal	binario natural	BCD
1	0001	0001
2	0010	0010
3	0101	0101
...	...	...
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010

BCD Aiken

Otro tipo de código BCD es el Aiken donde para generar esta codificación se usan una ponderación de bits diferente a la natural; si en la natural utilizábamos 8 4 2 1, en la Aiken la ponderación es 2 4 2 1. Es un sistema empleado por algunas unidades dedicadas a realizar restas.

BCD exceso a 3

El tipo BCD en exceso a 3 se obtiene sumando 3 al BCD natural. Se trata de una codificación no ponderada, en este caso el número no se obtiene por el peso de cada bit. Se trata de un sistema usado por unidades de suma.

Decimal	Aiken	Exceso 3	BCD Natural
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0010
3	0011	0110	0011
4	0100	0111	0100
5	1011	1000	0101
6	1100	1001	0110
7	1101	1010	0111
8	1110	1011	1000
9	1111	1100	1001

Para la conversión de binario decimal a otros sistemas de representación podemos utilizar circuitos integrados ya preparados para realizarte tipo de conversiones que nos facilitarán la tarea de transformación.

Gray

El último tipo de representación que se verá es el Gray, muy usado en robótica y aplicaciones industriales. Su propiedad es que entre la representación de dos números consecutivos nunca hay más de un bit de variación lo que lo hace especialmente interesante para evitar errores de codificación. A las codificaciones en las que sólo varía un bit entre dos números consecutivos se las denomina codificaciones progresivas o continuas.

Decimal	Gray
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Si nos fijamos, entre el 15 y el 0 también vuelve a haber un solo bit de diferencia, por lo que este código además es cíclico, por lo que podemos encontrarlos en discos de codificación como el de la figura.

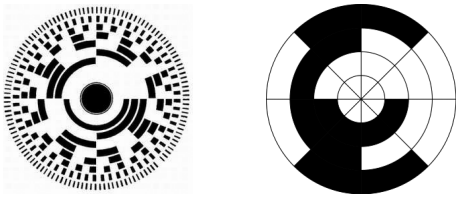


Figura B.1. Discos de codificación Gray.

Estos discos de codificación se pueden encontrar simplemente pintados (con lo que hay que realizar reconocimiento óptico), con contactos eléctricos o con unos agujeros por ejemplo en los 0, de modo que usados junto con diodos emisores de luz y fotoreceptores permiten saber la posición en la que se encuentran sabiendo que fotoreceptores reciben luz, y en caso de que alguno de ellos falle, siempre tendremos una sola posición de error, con lo cual se minimiza el impacto del mismo.

Sistema Hexadecimal

El sistema hexadecimal utiliza 16 símbolos para representar las cantidades, de ahí su denominación de base 16 o hexadecimal. Funciona de una manera semejante a la base decimal, solo que al llegar al 9, el siguiente número se representa mediante la letra A y así hasta que se acaban los 16 caracteres y se utilizan entonces dos cifras para la siguiente representación numérica:

Decimal	Hexadecimal
0	0
1	1
2	2
...	...
9	9
10	A
11	B
12	C
13	D
14	E

Decimal	Hexadecimal
15	F
16	10
17	11
...	...
196	C4

Resulta interesante que precisamente mediante un *nibble* se pueden representar exactamente 16 combinaciones distintas, es decir un byte lo podemos representar con dos símbolos.

```
10110101 = B5 = 181
```

Las conversiones entre binario y hexadecimal y viceversa se realizan de forma muy rápida teniendo en cuenta la tabla siguiente:

Hexadecimal	Binario
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Para realizar la conversión de hexadecimal a binario simplemente hay que ir transformando cada símbolo hexadecimal por su correspondiente binario por ejemplo:

```
B  = 1011
5  = 0101
B5 = 10110101
```

Para la conversión de binario a hexadecimal se realiza de manera inversa, se realizan agrupaciones de 4 bits comenzando por la derecha y se traduce de binario a hexadecimal según la tabla anterior.

```
11010100 → 1101   |   0100
1101 = C
0100 = 4
11010100 = C4
```

En caso de querer hacer transformaciones entre base decimal y hexadecimal el procedimiento no es tan directo.
Para proceder al cambio de hexadecimal a decimal, nos vamos a basar en el polinomio:

$$N = A_n*B^n + A_{(n-1)}*B^{(n-1)} + A_{(n-2)}*B^{(n-2)} + \ldots + A_2*B^2 + A_1*B^1 + A_0*B^0$$

y actuaremos de modo semejante a como se hizo en base binaria pero teniendo en cuenta que en este caso la base es 16 y los pesos son de la magnitud de 16^n. Si quisiéramos convertir el número hexadecimal D5C a decimal, tendríamos:

$$N = D * 16^2 + 5 * 16^1 + C * 16^0 = 13 * 16^2 + 5 * 16^1 + 12 * 6^0 = 3420$$

Para la conversión de decimal a hexadecimal, seguiremos el mismo procedimiento que como hicimos en binario, dividiendo por la base que en este caso es 16 y quedándonos con los restos o hasta que no se pueda dividir más; para formar el número decimal lo componemos de derecha a izquierda con todos los restos y con el número no divisible por la base. Para convertir el número 2016 a hexadecimal se procedería:

```
2016 / 16 = 126 resto 0
126  / 16 = 7   resto 14 (en hexadecimal E)
2016 en hexadecimal 7E0
```

Por ejemplo para el número 43917 sería:

```
43917 / 16 = 2744 resto 13 (en hexadecimal D)
2744  / 16 = 171  resto 8
171   / 16 = 10 (en hexadecimal A) resto 11 (en hexadecimal B)
43917 en hexadecimal AB8D
```





Índice alfabético

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z