ARDUINO

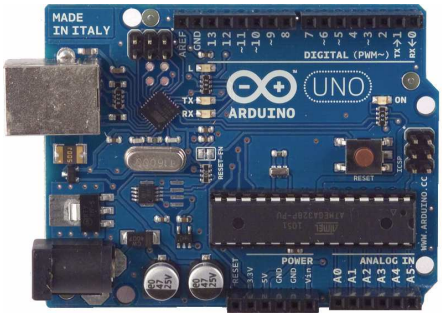
INDICE

ARDUINO

1.- INTRODUCCIÓN

Arduino es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo integrado (IDE), diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares, que pueden abarcar desde sencillas aplicaciones electrónicas domésticas hasta proyectos más elaborados para la industria. Su principal ventaja es su facilidad de programación, al alcance de cualquiera. Para encontrar toda la información que necesites sobre Arduino en la web http://www.arduino.cc/es/.

2.- EL HARDWARE DE LA PLACA ARDUINO



Las placas Ardukiino que usaremos habitualmente son del tipo UNO, la cual incorpora el chip ATmega328.

Tienen 14 entradas/salidas digitales, 6 entradas analógicas, entradas de alimentación, conexión USB para conectar al ordenador, conector de alimentación y un botón de Reset.

2.1. Alimentación

Las placas pueden ser alimentadas por medio de la conexión USB o con una fuente de alimentación externa de entre 6 a 20 V, aunque el rango recomendado es de 7 a 12 V.

Las fuentes de alimentación externas (no-USB) se pueden conectar usando un conector macho de 2.1 mm con centro positivo en el conector hembra de la placa, o mediante cables a los pines **Gnd** y **Vin** en los conectores de alimentación (POWER)

2.2. Entradas y Salidas

Cada uno de los **14 pines digitales** (de 0 a 13) pueden utilizarse como entradas o como salidas usando las funciones pinMode(), digitalWrite() y digitalRead(). Las E/S operan a 5 V. Cada pin puede proporcionar o recibir una intensidad máxima de 40 mA.

Los pines **3, 5, 6, 9, 10,** y **11** proporcionan una salida PWM (modulación por anchura de pulsos) de 8 bits de resolución (valores de 0 a 255) mediante la función **analogWrite()**. Esto permite sacar por estos pines, funcionando como salidas, valores de tensión comprendidos entre 0 y 5 voltios.

El pin digital 13 lleva conectado un LED integrado en la propia placa. Se encenderá cuando dicho pin se configura como salida y adopte un valor HIGH.; con valor LOW se apaga.

La placa tiene **6 entradas analógicas**, y cada una de ellas proporciona una resolución de 10 bits (1024 valores). Estas entradas también pueden funcionar como pines digitales.

2.3. Comunicaciones

La placa Arduino proporciona comunicación vía serie a través de los pines digitales **0** (RX) y **1** (TX). Un chip integrado en la placa canaliza esta comunicación serie a través del puerto USB. El software de Arduino incluye un **monitor de puerto serie** que *permite enviar y recibir información textual hacia y desde la placa Arduino*. Los leds RX y TX de la placa parpadearán cuando se detecte comunicación transmitida a través de la conexión USB.

2.4. Programación

La placa Arduino se puede programar a través del IDE (Entorno de Desarrollo Integrado) de Arduino. Primero hay que decirle al IDE nuestro tipo de placa. Para ello, selecciona "Arduino Uno" del menu **Herramientas > Tarjeta**. También hay que decirle al IDE en qué puerto USB hemos conectado la placa. Lo haremos a través del menú **Herramientas > Puerto Serial**. Aparecen una serie de puertos denominados COMx, donde x es un número. Para saber cuál es, desconecta y conecta el cable USB del ordenador, y el que desaparezca y vuelva a aparecer es nuestro puerto.

3. EL ENTORNO DE DESARROLLO PARA ARDUINO

El Entorno de Desarrollo Arduino permite la conexión del ordenador con la placa para cargar los programas y comunicarse con ella. A través de los menús de la barra de menús se puede acceder a todas las opciones del programa.

El programa o "*sketch*" se escribe en el editor de texto (se puede cortar, copiar, pegar, etc.,). En el área de mensajes se muestra información mientras se cargan los programas y también muestra los errores. La barra de herramientas contiene los comandos más habituales:

Verificar
Chequea el código en busca de errores.

Cargar
Compila el código y lo vuelca en la placa Arduino.

Nuevo
Crea un nuevo *sketch*.

Abrir
Presenta un menú de todos los programas de su "sketchbook" (*librería de sketch*). Un click sobre uno de ellos lo abrirá en la ventana actual.

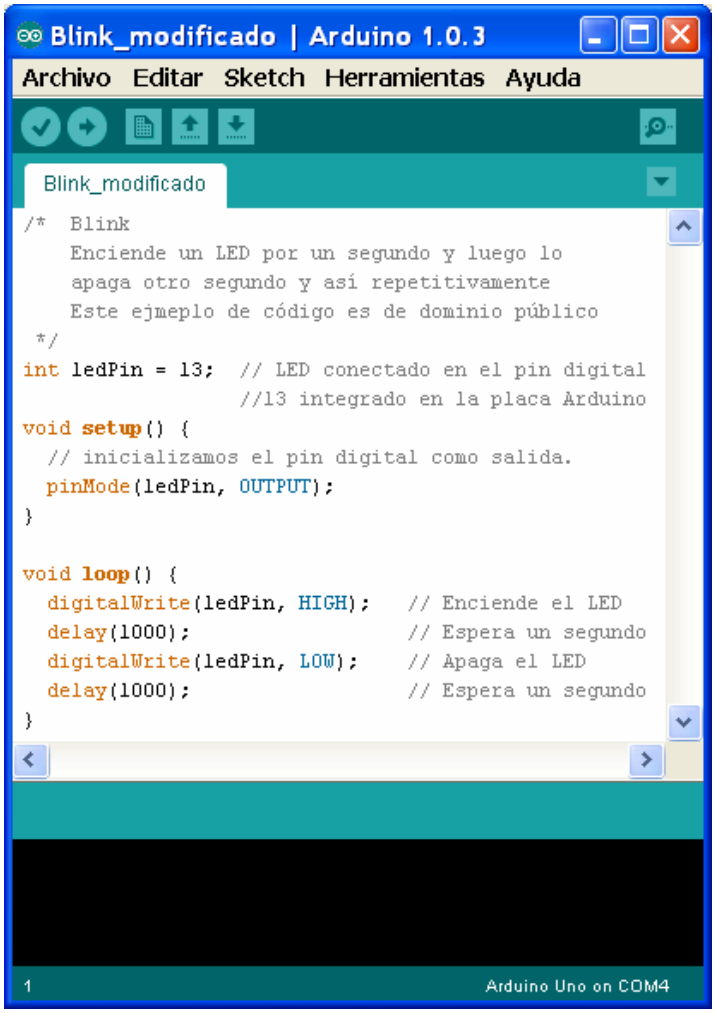
Guardar
Guarda el programa o *sketch*.

Monitor Serial
Inicia la monitorización del puerto serie.

Menú Gestor de pestañas
Permite gestionar pestañas pertenecientes a un mismo programa. Muy útil en programas largos. Podemos colocar las funciones en pestañas separadas. Al guardar, se guardan por separado en la misma carpeta con el nombre de la pestaña principal, que es con la que hay que abrir el archivo posteriormente para que se abran todas las pestañas.

4. NUESTRO PRIMER SKETCH

En la figura se muestra el sketch Blink_modificado.



Sketch es el nombre que usa Arduino para referirse a un programa.

4.1. Comentarios

Las primeras líneas del sketch, las cuales se encuentran entre /* y */, son comentarios y son ignorados por el compilador de Arduino. Aparecen en gris.

Otra forma de incluir comentarios cortos es mediante el //. Todo lo que sigue a la doble barra hasta el final de línea se considera comentario. Por ejemplo, en la línea:

```
int ledPin = 13;   // LED conectado en …
```

Lo primero es una instrucción, pero lo que sigue a // es un comentario.

4.2. Variables

Una variable es un lugar donde se almacena un dato. Posee un nombre, un tipo y un valor. Por ejemplo, en la línea:

```
int ledPin = 13;
```

se está declarando una variable de nombre *ledPin*, de tipo *int* (entero) y se le está asignando el valor *13*. Más adelante, podemos hacer referencia a esta variable por su nombre, para acceder a su valor, para utilizarlo o para modificarlo. Por ejemplo, en la instrucción siguiente se le está pasando el valor 13 al primer parámetro de la función pinMode():

```
pinMode(ledPin, OUTPUT);
```

Podríamos haber pasado a la función directamente el valor 13, es decir:

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

La ventaja de usar una variable en este caso es que sólo necesitas especificar su valor una única vez. Si más tarde decides cambiar, por ejemplo, el valor 13 por el 12, sólo tienes que cambiarlo una vez, concretamente en la línea del código donde se da a ledPin el valor 13.

Otra ventaja de una variable con respecto a un valor, como un número, es que puedes cambiar el valor de la variable mediante un operador de asignación (signo “=”). Por ejemplo, la sentencia:

`ledPin = 12;` cambiará el valor de la variable ledPin a 12.

Ten en cuenta que tienes que declarar una variable antes de asignarle un valor. Si se incluye la instrucción anterior en un programa sin la previa declaración del tipo de la variable, obtendrás un mensaje de error del estilo: "*error: ledPin was not declared in this scope*" ("Error: ledPin no ha sido declarado en este ámbito").

Cuando se asigna una variable a otra, estamos haciendo una copia de su valor en la posición de memoria asociada a la otra variable. Por ejemplo, con las siguientes instrucciones:

```
int ledPin = 13;
int pin2 = ledPin;
ledPin = 12;
```

declaramos y iniciamos *ledPin* con el valor 13; declaramos e iniciamos *pin2* con el valor contenido en *ledPin* (que en ese momento es 13); cambiamos el valor 13 de *ledPin* por 12. Por tanto, en *pin2* sigue estando en valor 13.

4.3. Las funciones setup() y loop()

Una función es una porción de código que puede ser utilizado desde cualquier lugar del sketch.

Las funciones *setup()* y *loop()* son dos funciones especiales que tienen que aparecer en todos los sketch de Arduino. Tienen que ser definidas por el usuario.

- La función `setup()` es llamada una sola vez, cuando comienza el sketch. Es un buen lugar para realizar tareas de configuración, como definir los pines o inicializar bibliotecas.
- La función `loop()` se llama una y otra vez de forma cíclica y es el corazón de la mayoría de los sketches.

Por ejemplo, aquí está la definición de la función `setup()` en el ejemplo Blink:

```
void setup()
{
  // Inicializamos el pin digital como salida
  pinMode(ledPin, OUTPUT);
}
```

La primera línea nos indica el nombre de la función, en este caso "setup". El texto que hay antes del nombre especifica el tipo de devolución (en este caso “*void*” significa que no devuelve nada). Entre los paréntesis se indican los parametros de la función (en este caso, la función *setup* no tiene parámetros). El codigo entre { y } es conocido como cuerpo de la función, o sea, lo que la función realiza.

¡Importante!: Hay que incluir ambas funciones en todo sketch, aún cuando no hagan nada.

4.4. Las funciones pinMode(), digitalWrite() y delay()

Arduino tiene ya definidas muchas funciones que podemos usarlas directamente sin tener que definirlas, simplemente invocando su nombre y proporcionándoles los parámetros que les corresponda. Veamos las tres que aparecen en el sketch:

- La función pinMode() configura un pin como entrada o salida. Para utilizarla, se le pasan sus dos parámetros: el primero es el número del pin que se va a configurar y el segundo es una constante. Si la constante es INPUT el pin se define como una entrada; si la constante es OUTPUT el pin se define como una salida. Cuando está configurado como entrada, un pin puede detectar el estado de un sensor, como un pulsador. Como salida, puede manejar un actuador, como un LED.
- La función digitalWrite() envía un valor a un pin. Por ejemplo, la línea:

  ```
  digitalWrite(ledPin, HIGH);
  ```

  asigna a la salida ledPin (pin 13 en el ejemplo Blink) el valor HIGH, ó 5 V. Enviando LOW a un pin lo conecta a tierra, ó 0 volts.
- La función delay() hace a Arduino esperar un tiempo dado por el número especificado de milisegundos dentro del paréntesis antes de continuar con la siguiente línea. Por ejemplo, la siguiente línea crea un retraso de un segundo:

  ```
  delay(1000);
  ```

5. LA ESTRUCTURA: FUNCIONES setup() Y loop()

La estructura básica del lenguaje de programación Arduino se organiza en, al menos, dos funciones que encierran bloques de declaraciones o sentencias: **void setup()** y **void loop()**. **Ambas funciones deben aparecer en todo programa, aún cuando estén vacías**.

La función **setup()** se ejecuta al inicio del programa. Se usa para inicializar variables, asignar los modos (INPUT o OUTPUT) de los pines, inicializar la comunicación serie, etc. La función *setup()* sólo se ejecutará una vez, después de cada vez que se conecte la placa Arduino a la fuente de alimentación, o cuando se pulse el botón Reset de la placa.

La función **loop()** ejecuta un bloque de código de forma cíclica continuamente.

Ejemplo

```
int buttonPin = 3;

/* Setup inicializa la configuración de la comunicación serie y el modo del pin indicado por buttonPin */

void setup()
 {
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
 }
// La función loop() comprueba el estado del pin dado por buttonPin cada 1 segundo,
// y envía un carácter por la salida serie indicando su estado

void loop ()
 {
  if(digitalRead(buttonPin) == HIGH) Serial.write('H');
  else Serial.write('L');
  delay(1000);
```

```
}
```

6. ELEMENTOS DE SINTAXIS

6.1. ; (Punto y coma)

Se utiliza para finalizar una declaración (toda declaración debe terminar en ;) y para separar los elementos de un bucle *for* (se verá más adelante).

Ejemplo

```
int x = 13;
```

Advertencia: Olvidarse de finalizar una instrucción con “;” dará lugar a un error del compilador. Si surge un error del compilador aparentemente ilógico, una de las primeras cosas a comprobar es la falta de un “;” en las inmediaciones de la línea en la que el compilador indicó el error.

6.2. { } (Las llaves)

Las llaves se utilizan en diversas contrucciones. La principal aplicación es definir el comienzo y el final del bloque de declaraciones de una función. También se utilizan en los bucles (while, do...while y for) y en las sentencias condicionales (if, if...else).

A una llave de apertura "{" debe corresponderle siempre una llave de cierre "}". O sea, deben estar “emparejadas”. El entorno de Arduino comprueba el emparejamiento de las llaves. Sólo hay que seleccionar una llave, o hacer clic en el punto de inserción inmediatamente después de una llave, y su compañera se resaltará.

Para no olvidar una llave de cierre es buena práctica de programación escribir la llave de cierre inmediatamente después de escribir la llave de apertura. A continuación, se insertan uno o varios renglones entre las llaves y se empieza a escribir el código.

Llaves desemparejadas o colocadas en lugares equivocados dan lugar a errores del compilador misteriosos, que pueden ser difíciles de rastrear en un programa grande.

6.3. Comentarios

Los comentarios son líneas que se utilizan para informar o aclarar el funcionamiento del programa. Son ignorados por el compilador, y no se exportan al procesador, por lo que no ocupan espacio en la memoria de Arduino.

Hay dos formas diferentes de introducir comentarios:

- Comentario de una sola línea: empiezan con // (dos barras inclinadas) y terminan al final de la línea.
- Bloque de comentario (o multilínea): empiezan con /* y terminan con */. Pueden abarcar varias líneas.

Ejemplo

```
x = 5;  // Esto es un comentario de una línea. Todo lo que va tras la doble barra hasta el final  de la línea
```

```
/* Esto es un comentario de varias líneas - se usa para comentar bloques enteros de código
 if (gwb == 0) {    // un comentario de una sola línea puede ir  dentro de un comentario multilínea
x = 3;  /* pero no otro comentario de varias líneas –  o sea, esto no es válido */
 }
 // No olvide el "cierre" del comentario - que tiene que ser equilibrado, como las llaves.
 */
```

Consejo: Cuando se producen errores al experimentar con código durante la elaboración de un programa, una forma habitual de ir aislando el código que puede ser erróneo es "meter las partes sospechosas dentro de un comentario". Este método permite dejar inoperativo esta parte del código (pues el compilador lo ignorará) pero sin borrarla.

7. OPERADORES ARITMÉTICOS

7.1. = Operador de asignación (un único signo igual)

El ***operador de asignación*** ("=") le dice al microcontrolador que evalúe el valor o expresión del lado derecho del signo "=" y lo almacene en la variable indicada a la izquierda de dicho signo.

Ejemplo

```
int senVal;                 // declara una variable entera llamada senVal
senVal = analogRead(0);     // almacena el voltaje de entrada (digitalizada) en el pin analógico 0 en senVal
```

Consejos de programación

- El tipo de la variable en el lado izquierdo del "=" tiene que ser capaz de contener el valor que se desea almacenar en ella. Si no es lo suficientemente grande el valor almacenado en la variable será incorrecto.
- No se debe confundir el operador de asignación (=) con el operador de comparación (==), que evalúa si dos expresiones son iguales.

7.2. (+ , – , * , /) Suma, resta, multiplicación y división

La operación se lleva a cabo utilizando el tipo de datos de los operandos, por lo que, por ejemplo, 9 / 4 da 2 (no 2.25) dado que 9 y 4 son enteros. Si los operandos son de tipos diferentes, se utiliza el tipo "más grande" para el cálculo.

Ejemplos

```
y = y + 3;
x = x – 7;
i = j * 6;
r = r / 5;
```

Consejos de programación:

- Las *constantes enteras* (números usados directamente) son por defecto de tipo *int*, por lo que algunos cálculos con constantes pueden provocar *desbordamiento* (por ejemplo, 60 * 1000 dará un resultado negativo, al ser 60000 superior a 32767, que es lo máximo que puede almacenar una constante o una vartiable tipo *int*).
- Hay que elegir tamaños de variables lo suficientemente grandes como para alojar el mayor resultado de los cálculos.
- Para las operaciones matemáticas que requieren fracciones, se deben utilizar las variables ***float***, aunque tienen el inconveniente de su gran tamaño (ocupan más memoria).

7.3. % (Módulo, o resto)

Calcula y devuelve el resto de dividir un entero por otro. No funciona con datos float.

Ejemplos

```
x = 7 % 5;   // x ahora contiene 2
 x = 5 % 5;   // x ahora contiene 0
 x = 4 % 5;   // x ahora contiene 4
```

8. OPERADORES COMPUESTOS

8.1. ++ (Incremento) / – – (decremento)

Incrementa o decrementa una variable. Devuelve su valor original o el recién modificado dependiendo de la posición que ocupen los signos con respecto al nombre de la variable.

Sintaxis

```
x++;   // incremento de x en 1 y devuelve el valor antiguo de x
++x;   // incremento de x en 1 y devuelve el nuevo valor de x

x– –;    // decremento de x en 1 y devuelve el valor antiguo de x
– –x;   // decremento de x en 1 y devuelve el nuevo valor de x
```

Ejemplos

```
x = 2;
y = ++x; // x ahora contiene 3, y contiene 3
y = x– –; // x contiene 2 de nuevo, y todavía contiene 3
```

9. CONSTANTES

Las constantes son variables predefinidas en el lenguaje de Arduino.

9.1. Definición de niveles lógicos, true y false (constantes booleanas)

true y **false** se utilizan para representar la verdad y la falsedad en el lenguaje de Arduino.

- **false** (falso) se define como 0 (cero).
- **true** (verdadero) es cualquier número distinto de 0. O sea, 1, – 1, 2 y –200 todos se definen como *true*, en un sentido booleano.

Nota: Téngase en cuenta que las constantes ***true*** y ***false*** se escriben en minúscula a diferencia de HIGH, LOW, INPUT y OUTPUT, que veremos a continuación, que se escriben en mayúsculas.

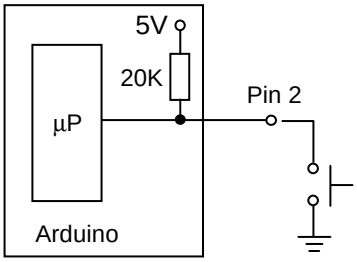
9.2. Definición de los pines digitales, INPUT y OUTPUT

Los pines digitales se pueden utilizar como **entrada** (**INPUT**) o como **salida** (**OUTPUT**). El cambio de un pin de entrada a salida o viceversa se realiza con **pinMode()**. Por defecto son de entrada, por lo que no tienen que ser explícitamente declarados como entradas con *pinMode()*.

Existe una definición de pin que es **INPUT_PULLUP**; en este caso, el pin se define como **entrada** y además **se activa su resistencia interna *pull up*** (de unos 20 K). Las resistencias *pull up* son resistencias conectadas entre las entradas y +5V.

Ejemplo:

```
pinMode(2, INPUT_PULLUP);
```



En este caso, cuando el pulsador no está pulsado, en el pin 2 se leerá, con digitalRead(2), un HIGH; cuando está pulsado se leerá un LOW;

Los pines configurados como salidas pueden ser dañados o destruidos si se ponen en cortocircuito a tierra cuando el pin está en valor HIGH o a la alimentación de 5 V cuando están en nivel LOW. Por esta razón es conveniente conectar los pines de salida a otros dispositivos con resistencias de 470Ω ó 1K, limitando la corriente máxima.

9.3. Definición de niveles de pin, HIGH y LOW

Al leer o escribir en un pin digital sólo son posibles dos valores: HIGH y LOW.

- **HIGH**

  Cuando un pin se configura como entrada (INPUT) con pinMode() y se lee con digitalRead(), el microcontrolador devuelve HIGH si en el pin hay un voltaje de **3 V o más**.

  Cuando un pin se configura como salida (OUTPUT) con pinMode() y se pone a HIGH con digitalWrite(), el pin se encontrará a **5 V**.

- **LOW**

  Cuando un pin se configura como entrada (INPUT) con pinMode() y se lee con digitalRead(), el microcontrolador devuelve LOW si en el pin hay un voltaje de **2 V o menos**.

  Cuando un pin se configura como salida (OUTPUT) con pinMode() y se pone a LOW con digitalWrite(), el pin se encontrará a **0 V**.

10. TIPOS DE DATOS

10.1. void

La palabra clave ***void*** (vacío) se utiliza sólo en las declaraciones de funciones. Indica que la función no retorna ninguna información a la función desde la que se ha llamado. Por ejemplo, las funciones setup() y loop() realizan acciones pero no devuelven información al programa principal.

10.2. boolean

Un valor **booleano** contiene uno de dos valores, **true** o **false**. Una variable booleana ocupa sólo un byte de memoria.

Ejemplos

```
boolean marcha = false;  // crea la variable booleana llamada marcha y le asinga el valor false
running = ¡marcha;     // cambia la variable marcha de estado
```

10.3. char

El tipo de datos ***char*** ocupa 1 byte de memoria y almacena un valor de carácter. Los caracteres se escriben entre comillas simples, como ‘A’ (para cadenas de caracteres se usan comillas dobles, como "ABC").

Ejemplo

```
char miletra = 'A';
```

10.4. int

Los enteros son el principal tipo de datos para el almacenamiento de números, y almacenan un valor de 2 bytes. Esto supone un rango desde -32.768 a 32.767 (valor mínimo de –2^15 y un valor máximo de (2^15) –1).

Ejemplo

```
int ledPin = 13;
```

Advertencia

Cuando las variables superan su capacidad máxima, éstas se desbordan y vuelven de nuevo a su capacidad mínima. Hay que tener cuidado pues esto dará lugar a errores o comportamientos extraños.

10.5. unsigned int

Unsigned int (enteros sin signo) almacenan un valor de 2 bytes. Sin embargo, sólo almacenan los valores positivos, permitiendo un rango útil de 0 a 65.535 (2 ^16) – 1).

Ejemplo

```
unsigned int ledPin = 13;
```

10.6. long

Las variables ***long*** son variables de tamaño extendido para el almacenamiento de números enteros, y almacenan 32 bits (4 bytes), entre –2.147.483.648 y 2.147.483.647.

Ejemplo

```
long velocidadVar = 186000L; // La 'L' hace que la constante esté en formato long
```

10.7. unsigned long

unsigned long son variables que almacenan 32 bits (4 bytes), pero a diferencia de los *long* estándar, los *unsigned long* no almacenan los números negativos, por lo que su rango está entre 0 y 4.294.967.295 (2^32 – 1).

Ejemplo

```
unsigned long tiempo;

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

void loop ()
{
  Serial.print("Hora:");
  tiempo = millis();   //Asigna a la variable tiempo el tiempo transcurrido en ms desde el inicio del programa
  Serial.println (tiempo);    // Imprime el tiempo en milisegundos desde inicio del programa
    delay (1000);               // Espera un segundo a fin de no enviar cantidades masivas de datos
}
```

10.8. float

Es el tipo de datos para números de punto flotante (número decimal). Los números de punto flotante se utilizan para aproximar valores analógicos porque tienen una mayor resolución que los enteros. El valor de las variables tipo ***float*** puede estar en el rango de –3.4028235E+38 a 3.4028235E+38. Se almacenan como 32 bits (4 bytes) de información.

Los números de punto flotante no son exactos, y pueden producir resultados extraños en las comparaciones. Por ejemplo, 6.0 / 3.0 puede no ser igual a 2.0. En su lugar, se debe comprobar que el valor absoluto de la diferencia entre el resultado y 2 es menor que una pequeña cantidad.

Ejemplo

```
float mivariable;
float Calibrador = 1.117;
```

Ejemplo de código

```
int x;
int y;
float z;
```

```
x = 1;
y = x / 2; // y ahora vale 0 (la parte entera de la operación, los datos int no pueden contener decimales)
z = (float)x / 2.0;       // z ahora contiene 0.5 (se tiene que utilizar 2.0, no 2)
```

10.9. Arrays (matrices)

Una matriz (array) es un conjunto de variables a las que se accede con un número de índice.

a) Crear (Declarar) una matriz

Todos los métodos siguientes son formas válidas para crear (declarar) una matriz.

```
int mimatriz[5];
int Pines[] = {2, 4, 8, 3, 6};
int misValores[5] = {2, 4, -8, 3, 2};
```

Posibilidades de declaración de matrices:

- Se puede declarar una matriz sin inicializarla, como la matriz *mimatriz*.
- En la segunda línea se declara la matriz *Pines* sin establecer explícitamente su tamaño. El compilador cuenta el número de elementos y crea una matriz del tamaño adecuado.
- Se puede inicializar y al mismo tiempo asignar el tamaño de la matriz, como en *misValores*.

b) Acceder a una matriz

Las matrices son **cero-indexadas,** es decir, al refererise a una matriz, el primer elemento de la matriz está en el índice 0, por lo tanto, misValores[0] será igual a 2, misValores[1] será 4 y así sucesivamente.

Por esta razón, hay que tener cuidado en el acceso a las matrices. Si se accede más allá del final de una matriz (usando un número de índice mayor que el tamaño declarado – 1) se leerá en una posición de la memoria que se utiliza para otros fines, dando lugar a errores. Igualmente, escribir en lugares erróneos de la memoria, “machacando” lo que allí haya dará lugar a un mal funcionamiento del programa. Este tipo de errores también son difíciles de localizar.

c) Para asignar un valor a una matriz

```
misValores[0] = 10;
```

d) Para recuperar un valor de una matriz

```
x = misValores[4];
```

e) Matrices y bucles for

Las matrices se manipulan muchas veces dentro de los bucles ***for***, donde se utiliza el contador del bucle como el índice de cada elemento de la matriz. Por ejemplo, para imprimir los elementos de una matriz a través del puerto serie, se podría hacer algo como esto:

```
int i;
for (i = 0; i <5; i = i + 1) {
  Serial.println (Pines[i]);
}
```

Ejemplo

```
// Knight Rider 2   El coche fantástico

int pinArray [] = {2, 3, 4, 5, 6, 7};
int contador = 0;
int temporizacion = 100;

void setup () {   // Hacemos todas las declaraciones a la vez
  for (contador = 0; contador <6; contador ++) {
    pinMode (pinArray [contador], OUTPUT);
  }
}
void loop () {
  for (contador = 0; contador <6; contador + +) {
   digitalWrite (pinArray [contador], HIGH);
   delay (temporizacion);
   digitalWrite (pinArray [contador], LOW);
   delay (temporizacion);
  }
  for (contador = 5; contador> = 0; contador --) {
   digitalWrite (pinArray [contador], HIGH);
   delay (temporizacion);
   digitalWrite (pinArray [contador], LOW);
   delay (temporizacion);
  }
}
```

11. FUNCIONES DE ENTRADA/SALIDA DIGITALES

11.1. pinMode(pin, modo)

Configura el pin especificado para comportarse en modo INPUT (entrada) o en modo OUTPUT (salida). También puede utilizar el modo INPUT_PULLUP (entrada con activación de la resistencia pull up del pin). Ver apartado 9.2 (Definición de los pines digitales, INPUT y OUTPUT).

pinMode() no devuelve nada. Véase la descripción de **los pines digitales** para más detalles.

Ejemplo

```
int ledPin = 13;  // LED conectado al pin digital 13
void setup ()
{
  pinMode (ledPin, OUTPUT);   // configura el pin digital como salida
}
void loop ()
{
  digitalWrite (ledPin, HIGH); // enciende el LED
  delay (1000);                // espera un segundo
```

```
  digitalWrite (ledPin, LOW); // apaga el LED
  delay (1000);                    // espera un segundo
}
```

Nota: Los pines de entrada analógica se pueden utilizar como pines digitales, referiéndose a ellos como A0, A1, etc. para distinguirlos de los digitales. En las funciones que sólo operan con los pines analógicos, como analogRead(), no hace falta poner la A delante del nº de pin. Por ejemplo:

```
pinMode (A0, OUTPUT);
digitalWrite (A0, HIGH);
x = analogRead(0);    // es equivalente a  x = analogRead(A0)
```

11.2. digitalWrite(pin, valor)

Escribe un valor HIGH o LOW en el pin digital especificado. No devuelve nada.

Si el pin se ha configurado como OUTPUT (salida) con pinMode(), su tensión se establece en 5V para HIGH y a 0V (tierra) para LOW.

Nota: el pin digital 13 es más difícil de usar como entrada digital porque tiene un LED y una resistencia fija incorporados a la placa. Para evitar errores es mejor utilizarlo sólo como salida.

Ejemplo

El mismo que el de la función pinMode() vista antes.

Nota Los pines de entrada analógica se puede utilizar como pines digitales, conocidos como A0, A1, etc. Ver nota y ejemplo en la función pinMode().

Nota: La función digitalWrite() también se puede utilizar para activar o desactivar las resistencias *pull up*. Así si tenemos un pin, por ejemplo el 2, definido como entrada (INPUT) con pinmode(2, INPUT) y ejecutamos digitalWrite(2, HIGH) activamos su resistencia *pull up*; si ejecutamos digitalWrite(2, LOW) desactivamos su resistencia *pull up*.

11.3. digitalRead(pin)

Lee el valor del pin digital especificado. Devuelve o HIGH o LOW .

Ejemplo

```
int ledPin = 13;  // LED conectado al pin digital 13
int inPin = 7;     // pulsador conectado al pin digital 7
int val = 0;       // variable para almacenar el valor leído

void setup()
{
  pinMode (ledPin, OUTPUT);  // establece el pin digital 13 como salida
  pinMode (inPin, INPUT);       // establece el pin digital 7 como entrada
}

void loop()
```

```
{
  val = digitalRead (inPin);  // lee el pin de entrada
  digitalWrite (ledPin, val);  // establece el LED al valor dado por el pulsador conectado al pin 7
}
```

Nota: Si el pin no está conectado a nada, digitalRead() puede devolver o HIGH o LOW de forma aleatoria, lo que puede dar lugar a errores. Conviene que las entradas estén conectadas a algo, como, por ejemplo, a tierra a través de una resistencia externa (resistencia *pull down*), para garantizar el valor LOW cuando no haya conectado nada. Otra opción es activar la resistencia interna *pull up*, con lo que garantizamos el valor HIGH cuado no haya nada conectado al pin.

12. FUNCIONES DE ENTRADA/SALIDA ANALÓGICA

12.1. analogRead(pin)

Lee el valor de tensión en el pin analógico especificado (0 a 5). La placa dispone de un convertidor analógico-digital que asignará a voltajes de entrada de entre 0 y 5 V valores enteros entre 0 y 1023. Por tanto, esta función devuelve un valor entero entre 0 y 1023.

Nota: Si el pin de entrada analógico no está conectado a nada, el valor devuelto por analogRead() va a fluctuar aleatoriamente en función de una serie de factores.

Ejemplo

```
// programa para monitorizar el valor de tensión en el terminal intermedio de un potenciómetro.
int miPinAnalog = 3;        // el terminal intermedio de un potenciómetro conectado al pin analógico 3
                            // los terminales externos conectados a tierra y +5 V
 int val = 0;          // variable para almacenar el valor leído por el conversor
 void setup ()
 {
   Serial.begin (9600);  // configuración de la comunicación serie
 }
 void loop ()
 {
   val = analogRead(miPinAnalog);        // lee el el valor de tensión en el pin de entrada
   Serial.println(val);                  // envía el valor leído vía serie
   delay(1000);
}
```

12.2. analogWrite(pin, valor)

Escribe un valor (entre 0 y 255) pseudo-analógico (onda PWM) en el pin digital especificado. Se puede utilizar para encender un LED con brillo variable o hacer girar un motor a varias velocidades. Después de llamar a **analogWrite(),** en el pin se generará una onda cuadrada constante con el ciclo de trabajo especificado (0 corresponde a siempre *"off"* y 255 a siempre "*on"*) hasta la siguiente llamada a **analogWrite()** (o una llamada a **digitalRead()** o **digitalWrite()** en el mismo pin).

En la Arduino UNO esta función funciona en los pines digitales 3, 5, 6, 9, 10 y 11. No es necesario llamar a pinMode() para establecer el pin como salida para poder usar la función analogWrite().

La función analogWrite() no tiene nada que ver con los pines analógicos o la función analogRead.

Ejemplo

Establece el brillo del LED proporcionalmente al valor de tensión leído en el potenciómetro.

```
int ledPin = 9;             // LED conectado al pin digital 9
int miPinAnalog = 3;   // potenciómetro conectado al pin analógico 3
int val = 0;                // variable para almacenar el valor leído
void setup ()
{
  pinMode (ledPin, OUTPUT);      // establece el pin como salida
}
void loop ()
{
  val = analogRead (miPinAnalog);     // lee el pin de entrada analógica
  analogWrite (ledPin, val/4);           // para escalar valores: los valores de analogRead van de 0 a 1023,
                                                  // los valores de analogWrite de 0 a 255
}
```

13. FUNCIONES DE COMUNICACIÓN SERIE

Se utilizan para la comunicación entre la placa Arduino y un ordenador u otros dispositivos. Las placas Arduino se comunican por los pines digitales **0** (RX) y **1** (TX), así como con el ordenador mediante la conexión USB. Por lo tanto, ***si utiliza estas funciones, no puede usar los pines 0 y 1 para entrada o salida digital***.

Se puede utilizar el monitor del puerto serie incorporado en el entorno de Arduino para comunicarse con la placa Arduino. Haga clic en el botón de monitor del puerto serie en la barra de herramientas y seleccione la misma velocidad utilizada en la llamada a Serial.begin().

13.1. Serial.begin(valor)

Establece la velocidad de transmisión de datos en bits por segundo (baudios) para la transmisión de datos serie. Para comunicarse con el ordenador, suele utilizarse 9600 baudios.

Ejemplo

```
void setup() {
    Serial.begin (9600);   // abre el puerto serie, establece la velocidad de datos a 9600 bps
 }
 void loop() {}
```

13.2. Serial.end()

Desactiva la comunicación serie, permitiendo a los pines 0 (RX) y 1 (TX) ser utilizados como entradas o salidas digitales. Para volver a habilitar la comunicación serie, se llama a *Serial.begin()*. La función Serial.end() no lleva ningún parámetro.

13.3. Serial.print(valor)

Imprime los datos al puerto serie como texto legible ASCII. Los datos *float* son impresos por defecto con dos decimales. Serial.print() no añade retorno de carro ni nueva línea.

Ejemplos:

- Serial.print (78); // imprime "78"
- Serial.print (1.23456); // imprime "1.23"
- Serial.print ('N'); // imprime "N"
- Serial.print ("Hola mundo."); // imprime "Hola mundo."
- Serial.print ("\t"); // imprime un tabulador

Un segundo parámetro opcional especifica el formato a usar. Para los números de punto flotante, este parámetro especifica el número de decimales a utilizar.

Ejemplos:

- Serial.print (1.23456, 0) imprime "1"
- Serial.print (1,23456, 2) imprime "1.23"
- Serial.print (1,23456, 4) imprime "1,2346"

13.4. Serial.println(valor)

Imprime los datos al puerto serie como texto legible ASCII seguido por carácter de retorno de carro ('\r') y un carácter de nueva línea ('\n'). Este comando tiene las mismas formas que *Serial.print()*, es decir, con sólo el argumento valor o con un segundo argumento opcional (formato).

Ejemplo

```
//  Lee una entrada analógica del pin analógico 0, imprime el valor por el puerto serie.
int analogValue = 0;   // variable para almacenar el valor analógico

void setup() {
   Serial.begin (9600);  // Abre el puerto serie a 9600 bps
}
void loop ) {
   analogValue = analogRead(0);  // Lee la entrada analógica en el pin 0
   Serial.println(analogValue);  // imprime el valor en código ASCII y pasa al siguiente renglón
   delay(200);  // Retardo de 200 milisegundos antes de la siguiente lectura.
}
```

13.5. Serial.available()

Devuelve el número de bytes (caracteres) disponibles para su lectura desde el puerto serie. Se refiere a datos que ya llegaron y que se almacenan en el búfer de recepción del puerto serie (que tiene 128 bytes).

No tiene ningún parámetro.

Ejemplo

```
int byteEntrante = 0; // variable para la entrada de datos serie
void setup () {
        Serial.begin (9600); // abre el puerto serie, establece la velocidad de datos a 9600 bps
 }
 void loop () {
        if (Serial.available()> 0) {  // Envía datos sólo cuando se reciben los datos:
                char byteEntrante = Serial.read(); // Lee el byte de entrada y lo conviente en carácter
                Serial.print ("he recibido: ");  // Indica lo que tienes:
                Serial.println (byteEntrante); //Imprime el carácter enviado
         }
 }
```

13.6. Serial.read()

Lee datos serie entrantes del puerto serie. Tras leerlo lo elimina. No lleva parámetros.

Devuelve el primer byte de datos serie de entrada disponible recibido por el puerto serie (ó –1 si no se dispone de datos), en tipo int.

Ejemplo

```
const int ledPin = 13;   // Conectaremos el LED en el pin 13 (usaremos el que lleva la placa)
int byteEntrante = 0;  // variable para almacenar los datos serie de entrada

void setup () {
        Serial.begin (9600); // abre el puerto serie, establece la velocidad de datos a 9600 bps
        pinMode (ledPin, OUTPUT);
 }
 void loop () {
        // Envía datos sólo cuando se reciben los datos:
        if (Serial.available()> 0) {
                char byteEntrante = Serial.read();    // Lee el byte de entrada y lo convierte en carácter.
                Serial.print ("he recibido: ");    // Indica lo que ha recibido:
                Serial.println (byteEntrante);
                if (byteEntrante == 'H') {    //si recibe H enciende el LED
                        digitalWrite (ledPin, HIGH);
                }
                if (byteEntrante == 'L') {     // si recibe L apaga el LED
                        digitalWrite (ledPin, LOW);
                }
         }
 }
```

13.7. Serial.flush()

Vacía el búfer de entrada de datos serie. Es decir, cualquier llamada a *Serial.read()* o *Serial.available()* devolverá sólo los datos recibidos después de la llamada más reciente a *Serial.flush()*. No devuelve nada. No lleva parámetros.

14. FUNCIONES DE TIEMPO

14.1. millis()

Devuelve el número de milisegundos transcurridos desde que la placa Arduino empezó a correr el programa actual. Este número se desbordará (volverá a cero), después de aproximadamente 50 días. El dato devuelto es de tipo unsigned long (rango de 0 a (2^32) – 1).

Ejemplo

```
unsigned long tiempo;

void setup () {
   Serial.begin(9600);
}
void loop () {
   Serial.print("Tiempo: ");
   tiempo = millis();
   Serial.println (tiempo); // Imprime el tiempo en milisegundos desde el inicio del programa
   delay(1000);   // Espera un segundo a fin de no enviar cantidades masivas de datos
 }
```

Advertencia: Téngase en cuenta que el dato que devuelve *millis()* es de tipo unsigned long, por lo que se pueden generar errores si se intenta hacer operaciones matemáticas con otros tipos de datos como enteros.

14.2. delay(valor)

Pausa el programa durante el tiempo (en milisegundos) especificado como parámetro. El dato dado como parámetro es de tipo unsigned long. Esta función no devuelve nada.

Ejemplo

```
int ledPin = 13; // LED conectado al pin digital 13

void setup () {
   pinMode (ledPin, OUTPUT);   // pone el pin digital como salida
}

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

Advertencia: Si bien muchos programas usan delay() para crear pausas cortas, como en el caso del parpadeo del LED del ejemplo inicial, o para **eliminar rebotes al conmutar interruptores**, el uso de delay() tiene desventajas ya que, miestras el programa está pausado, no hay lectura de los sensores, ni cálculos matemáticos, ni se pueden manipular los pines, etc. Un método alternativo para controlar el tiempo es el uso de la función millis().

Sin embargo, algunas cosas siguen funcionando mientras el programa está pausado por la función delay(). Las interrupciones siguen funcionando, los valores PWM (analogWrite) y los valores y estados de pin se mantienen.

Ejemplo: Blink sin retardo usando millis()

A veces es necesario hacer dos cosas a la vez. Por ejemplo, hacer parpadear un LED mientras se está atento a la pulsación de un botón en otra entrada. En este caso, no se puede usar delay(), ya que se para todo el programa mientras que el LED parpadea. El programa perdería la pulsación del botón si ocurre durante el tiempo de delay(), que es casi todo el tiempo. Este sketch muestra cómo hacer parpadear el LED sin usar delay(). Guarda la última vez que Arduino encendió o apagó el LED. Entonces, cada cierto tiempo, chequea si ha pasado un intervalo de tiempo determinado. Si es así, se cambia el LED de encendido a apagado o viceversa. El código utiliza la función millis() y no utiliza delay().

```
// Blink sin retardo
// Constantes que no van a cambiar:
const int ledPin =  13;      // se indica el pin donde va el LED, puede usarse el que incorpora la placa
                             // con el modificador const delante la variable se hace de “sólo lectura”
// Variables que van a cambiar:
int estadoLed = LOW;             // la variable estadoLed es usada para establecer el estado del LED

// las siguientes variables son long porque el tiempo es medido en milisegundos,
// por lo que se convertirá rápidamente en un número más grande de lo que se puede almacenar en un int.
long horaprevia = 0;          // almacenará la última vez que el LED se ha actualizado
long intervalo = 1000;        // intervalo de parpadeo (milisegundos)

void setup() {
  pinMode(ledPin, OUTPUT);     // establece el pin digital como salida
}

void loop() {
 // aquí es donde se pondría el código que debe estar en ejecución todo el tiempo.
 // Se chequea para ver si es el momento de conmutar el LED, es decir, si la diferencia entre la hora
 // actual y la última vez que se conmutó el LED es mayor que el intervalo de parpadeo:

 unsigned long horaactual = millis();

 if(horaactual - horaprevia > intervalo) {
       horaprevia = horaactual;          // actualiza la hora de la última vez que parpadeó el LED
       if (estadoLed == LOW) estadoLed = HIGH;    // Si el estado del LED está off lo pone on
       else estadoLed = LOW;                      // Si el estado del LED está on lo pone off
      digitalWrite(ledPin, estadoLed);            // establece el LED al estado indicado por estadoLed
 }
}
```

14.3. pulseIn (pin,valor)

La función adopta también la forma: pulseIn (pin, valor, timeout).

Lee un pulso (bien HIGH o LOW, en función de parámetro *valor*) en el pin dado por el parámetro *pin*. Devuelve la duración del pulso en microsegundos.

Por ejemplo, si valor es HIGH, la función espera a que el pin se ponga HIGH, cuando esto ocurre empieza a contar el tiempo hasta que el pin se ponga a LOW y entonces detiene el cómputo de tiempo y devuelve la duración del pulso en microsegundos.

Si no se produce ningún pulso en un determinado tiempo de espera (timeout) se interrumpe la medida y devuelve 0. Si no se especifica el parámetro timeout el valor por defecto es 1 segundo.

Los parámetros *pin* y *valor* son de tipo int, mientras que *timeout* es de tipo unsigned long.

Ejemplo

```
int pin = 7;
unsigned long duracion;

void setup () {
   pinMode (pin, INPUT);
}
void loop () {
   duracion = pulseIn (pin, HIGH);
}
```

15. ESTRUCTURAS DE CONTROL

15.1. if (condición)

if , comprueba si una determinada condición se cumple (como, por ejemplo, si una entrada posee un valor por encima de un determinado número) y, en caso afirmativo, ejecuta un bloque de código. El formato de la prueba if es:

```
if (condición)
 {
   // Bloque de código que debe ejecutarse si se cumple la condición
 }
```

El programa comprueba si la condición entre paréntesis es cierta (*true*). Si es así, el programa ejecuta la o las instrucciones dentro de las llaves. Si no, el programa se salta dicho código.

Las llaves se pueden omitir después de una sentencia if. Si se hace esto, la siguiente línea (definida por el ";") se convierte en la única instrucción afectada por la condición.

Todas las siguientes sentencias condicionales son correctas

```
if (x > 120) digitalWrite (ledPin, HIGH);
if (x > 120)
digitalWrite (ledPin, HIGH);
if (x > 120) {digitalWrite (ledPin, HIGH);}
```

```
if (x > 120) {
  digitalWrite (LEDpin1, HIGH);
  digitalWrite (LEDpin2, HIGH);
}
```

Las declaraciones que se evalúan dentro de los paréntesis requieren el uso de uno o más operadores de comparación y/o booleanos.

15.2. Operadores de comparación: == , ! = , < , > , <=, >=

x == y (x es igual a y)
x != y (x no es igual a y)
x < y (x es menor que y)
x > y (x es mayor que y)
x <= y (x es menor o igual a y)
x >= y (x es mayor o igual a y)

Advertencia: Cuidado con el uso accidental de un único signo igual dentro de la sentencia if. Por ejemplo, if (x = 10) no daría error al compilar pero haría algo diferente a lo que pretendemos.

15.3. Operadores booleanos

Estos pueden ser utilizados en el interior de la condición de una sentencia **if**.

- **&&** (AND lógico) se evalúa como *true* sólo si ambos operandos son *true*, por ejemplo:

  ```
  if (digitalRead(2) == HIGH  &&  digitalRead (3) == HIGH) {   // lee dos interruptores }
  ```

- **||** (OR lógico) se evalúa como *true* si alguno de los operandos es *true*, por ejemplo:

  ```
  if (x> 0 || y> 0) {// se ejecuta este bloque sin x ó y son mayores que 0 }
  ```

- **!** (NOT lógico) se evalúa como *true* si el operando es *false*, por ejemplo:

  ```
  if (!x) { //  se evalúa como true si x es false, es decir, si x es igual a 0 }
  ```

Ejemplo

```
if (a >= 10 && a <= 20) {} // verdadero si el valor de “a” está entre 10 y 20
```

15.4. If...else

If....else permite que se agrupen múltiples pruebas. Por ejemplo, una entrada analógica puede ser chequeada y tomarse una acción si se cumple una condición, y otra acción si no se cumple. Por ejemplo:

```
if (pinEntrada < 500)
{
  // Acción A
}
 else
 {
  // Acción B
```

```
}
```

A **else** puede proseguir otra prueba **if**, por lo que se pueden ejecutar sucesivamente múltiples pruebas mutuamente excluyentes.

A cada prueba proseguirá la siguiente siempre que su resultado sea falso. Cuando se encuentra una prueba verdadera, su bloque de código asociado se ejecuta, y entonces el programa salta a la línea siguiente a toda la construcción *if....else*. Si no existe ninguna prueba verdadera, el bloque **else** por defecto (establece el comportamiento por defecto) se ejecuta, si es que está presente (no es obligatorio). Si no hay ningún **else** declarado, no sucede nada.

```
if (pinEntrada <500)
{
  // Hacer cosa A
}
else if (pinEntrada > = 1000)
{
  // Hacer cosa B
}
else
{
  // Hacer cosa C, este bloque no es obligatorio
}
```

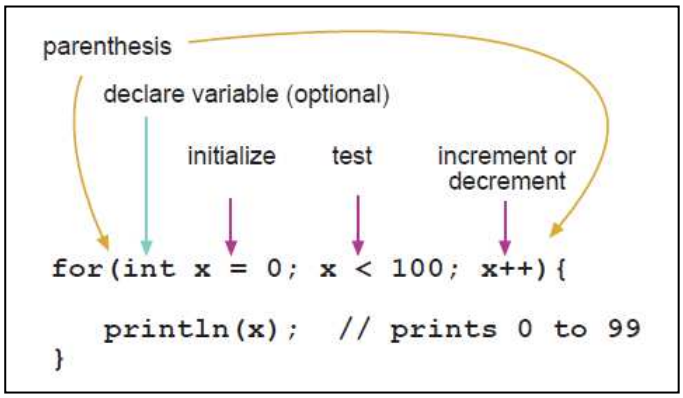
Otra forma de expresar la ramificación, es con la sentencia **switch... case** (ver más adelante).

15.5. for

La construcción ***for*** es usada para repetir un bloque de instrucciones (encerrado entre llaves) un número de veces. Se suele usar una variable como contador, la cual se modifica (incrementándola o decrementándola) en cada pasada del bucle. Cuando una condición basada en el valor de la variable contador deja de cumplirse, se sale del bucle.

Hay tres partes en la cabecera del bucle **for**:

```
for (inicializacion ; condicion ; modificación ) {
  //Sentencias;
}
```



La ***inicialización*** se realiza en primer lugar y solamente una vez. En cada pasada del bucle se comprueba la ***condición*****;** si es cierta, el bloque de instrucciones y la ***modificación*** se ejecutan; a continuación se prueba la ***condición*** de nuevo y así sucesivamente. Cuando la ***condición*** se vuelve falsa, el bucle termina.

Ejemplo

```
// Iluminanción progresiva de un LED conectado a un pin PWM
```

```
int PWMpin = 10;    // Ponemos un LED en serie con una resistencia de 470 Ω en el pin 10
void setup()
{
   // No es necesaria configuración pues el pin PWMpin se va a usar con analogWrite()
}
void loop()
{
  for (int i = 0; i <= 255; i++) {
    analogWrite(PWMpin, i);
    delay(10);
  }
}
```

Ejemplo

Usamos una multiplicación en el parámetro de *modificación*. El siguiente código genera: 2,3,4,6,9,13,19,28,42,63,94. (hay que tener en cuenta que al ser x tipo int se pierden los decimales).

```
for(int x = 2, x <100; x = x * 1.5) {
   Serial.println(x);
}
```

Ejemplo

Variación del brillo de un LED hacia arriba y hacia abajo con un bucle *for*:

```
int PWMpin = 10;    // LED en serie con una resistencia de 470 ohmios en el pin 10

void setup()
{
   // No es necesaria configuración
}

void loop ()
{
  int x = 1;
  for (int i = 0; i> – 1; i = i + x) {
    analogWrite (PWMpin, i);
    if (i == 255) x = –1;   // cambia de dirección en el pico
    delay (10);
  }
}
```

15.6. switch....case

La sentencia ***switch...case*** compara el valor de una variable con las etiquetas especificadas en las declaraciones ***case***. Cuando se encuentra una declaración *case* cuya etiqueta coincide con el valor de la variable, el código correspondiente a dicha sentencia *case* se ejecuta.

Se puede incluir, aunque no es obligatorio, una sección ***default***, que incluiría la porción de código que se ejecutaría si el valor de la variable no coincide con las etiquetas de ningún *case*.

La palabra clave ***break*** hace salir de la sentencia *switch*, y se suele utilizar al final de cada *case*.

Sintaxis

```
switch (var) {
  case etiqueta1:
    // Instrucciones que se ejecutan cuando var = etiqueta1
    break;
  case etiqueta2:
    // Instrucciones que se ejecutan cuando var = etiqueta2
    break;
  default:
    // Instrucciones que se ejecutan si el valor de var no coincide con ninguna etiqueta
    // default es opcional
}
```

15.7. while

El bucle ***while*** se repetirá indefinidamente, hasta que la expresión booleana dentro del paréntesis () se evalúe como *false*. Algo tiene que hacer cambiar la variable chequeada, o el bucle ***while*** nunca terminará.

Ejemplo

```
var = 0;
while (var < 200) {
  // Hacer algo repetitivo 200 veces
  var++;
}
```

15.8. do - while

El bucle ***do*** opera en la misma forma que el bucle ***while***, con la excepción de que la condición se comprueba al final del bucle, por lo que el bucle siempre *se ejecutará al menos una vez*.

Ejemplo

```
do
 {
   delay(50);                 // espera a que los sensores se estabilicen
   x = LeeSensores();         // función LeeSensores() creada por el usuario que comprueba los sensores
 } while (x < 100);           // si se cumple la condición se repite el bucle
```

15.9. break

break se utiliza para salir de un bucle (**for, while** o **do-while**), independientemente de que se cumpla la condición del bucle. También se utiliza para salir de una estructura ***switch...case*.**

Ejemplo

```
for (x = 0; x < 255; x++)
 {
    analogWrite (PWMpin, x);
    sens = analogRead (sensorPin);
    if (sens > umbral) {                // Este código va aumentando el valor en la salida PWMpin hasta que
      x = 0;                            // el valor leído en sensorPin llega al valor  establecido en la variable
      break;   // Sale del bucle for       umbral, una vez llega, la salida se mantiene en ese valor
    }
    delay (50);
 }
```

15.10. continue

La sentencia ***continue*** salta el resto de la iteración actual de un bucle (**for, while** o **do-while**). Continúa chequeando la expresión condicional del bucle, y procede con una nueva iteración.

Ejemplo

```
 for (x = 0; x <255; x++)
 {
    if (x > 40  &&  x < 120) {        // Con este código el valor colocado en la salida PWMpin va creciendo
       continue;                      //  desde 0 hasta llegar a 40, a continuación se produce un salto al valor
    }                                 //  200 y a continuación sigue subiendo hasta 254.
    analogWrite(PWMpin, x);
    delay(50);
 }
```

15.11. return

Termina una función y devuelve, si se desea, un valor desde dicha función a la función que la ha llamado. Una función puede no devolver nada.

Sintaxis

Hay dos formas válidas, sin devolver nada o devolviendo un valor.

```
  return;
  return valor;  // valor puede ser cualquier constante o variable
```

Ejemplo

Una función para comparar una entrada de sensor con un umbral.

```
 int ChequeaSensor() {
    if (analogRead (0) > 400) {
       return 1;
    else {
       return 0;
    }
 }
```

16. VARIABLES

Una variable es un modo de nombrar y almacenar un valor para su uso posterior por el programa.

a) Declarar variables

Antes de que se utilicen, todas las variables tienen que ser declaradas. Declarar una variable significa definir su tipo, y, opcionalmente, asignarle un valor inicial (inicialización de la variable).

```
int variableEntrada1;
int varCalibracion = 17; // declara la variable varCalibracion y establece su valor inicial en 17
```

b) Desbordamiento de variables

Cada tipo de variable lleva asociado un espacio en memoria para almacenar su valor. Por ejemplo, las variables de tipo int 2 bytes, las de tipo long o float 4 bytes, etc. Cuando las variables superan su capacidad máxima vuelven de nuevo a su capacidad mínima y viceversa. Esto se denomina *desbordamiento*.

```
int x;
x = 32.767;        // es el valor máximo que puede tener una variable tipo int
x = x + 1 // x ahora contiene –32,768  (se da la vuelta)
x = –32.768;
x = x – 1; // x contiene ahora 32.767 ( se da la vuelta en dirección contraria )
```

c) Utilización de variables

Una vez que las variables han sido declaradas, son utilizadas para hacerlas igual al valor que se desea almacenar en ellas mediante el operador de asignación (“=”). También se puede probar si su valor cumple una condición.

```
variableEntrada1 = 7;                    // establece la variable llamada variableEntrada1 a 7
variableEntrada2 = analogRead(2);        // establece variableEntrada2 al valor leído en el pin analógico 2
if (variableEntrada2 <100) variableEntrada2 = 100;
delay (variableEntrada2);    // se usa el valor de la variable como parámetro de entrada a la función delay
```

Nota sobre estilo: Se deben dar a las variables nombres descriptivos, a fin de que el código sea más fácil de entender. Se puede nombrar a una variable con cualquier palabra que no sea ya una de las *palabras clave* en Arduino.

d) Ámbito de las variables

Dependiendo del lugar donde se declara una variable en un programa podremos usar dicha variable en unas partes o en otras del programa. Existen dos tipos básicos de variables:

- **Variables globales:** se pueden utilizar y modificar en cualquier parte del programa. **Se declaran fuera de cualquier función**, normalmente en la parte superior del programa.

  Ejemplo:

```
int ledPin = 13;

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

void loop()
{
  digitalWrite(ledPin, HIGH);
}
```

  La variable ledPin es de tipo global, se puede utilizar tanto en la función setup() como en la función loop().

- **Variables locales:** se pueden utilizar sólo en una función, y se declaran dentro de dicha función.

  Ejemplo:

```
void setup()
{
  int ledPin = 13;
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, HIGH);
}
```

  Ahora la variable ledPin sólo podrá ser usada dentro de la función setup(). Si tratamos de utilizarla en otra función, como la función loop(), dará lugar a un error. Por ejemplo:

```
void loop()
{
  digitalWrite(ledPin, LOW); // incorrecto: ledPin no tiene ámbito aquí.
}
```

Cuando los programas empiezan a hacerse más grandes y más complejos, las variables locales son un medio útil para asegurarse de que sólo una función tiene acceso a sus propias variables. Esto evita errores de programación cuando una función modifica inadvertidamente las variables utilizadas por otra función. Además, si en alguna parte hay un error, será mucho más difícil de encontrar. Esta es la razón de que no se hagan globales todas las variables.

A veces también es útil declarar e inicializar una variable dentro de un bucle ***for***. Esto crea una variable que sólo se puede acceder desde el interior de las llaves del bucle for.

Ejemplo

```
int gPWMval;   // cualquier función verá y tendrá acceso a esta variable, es global
```

```
void setup ()
{
    // ...
}

void loop ()
{
  int i;              // "i" sólo es "visible" dentro de "loop", es local
  float f;           // "f" es sólo "visible" dentro de "loop", es local
     // ...

  for (int j = 0; j <100; j++) {
              // La variable j sólo es accesible dentro de las llaves del bucle for, es local
  }
}
```

16.1. static

La palabra clave ***static*** se utiliza para crear variables estáticas. Estas variables son visibles en una sola función o bloque de código, como las locales, pero tienen la particularidad de que no se crean y se destruyen cada vez que se llama a la función, como les ocurre a las variables locales normales, sino que su valor se guarda para las sucesivas llamadas.

Las variables declaradas como *static* sólo se crean y se inicializan la primera vez que se llama a la función o bloque de código en que han sido creadas.

Ejemplo

```
int fun_cambialado (int incre_lado) {
  static int lado = 0;              // variable estática para almacenar el valor del lado,
  lado = lado + incre_lado;    // declarada como static para que se almacene el valor de lado entre llamada y
                                    // llamada a la función, pero que otras funciones no puedan cambiar su valor
  return lado;
}
```

16.2. const

La palabra clave ***const*** es un *calificador de variable* que modifica el comportamiento de la variable, haciéndola de "sólo lectura*".* Esto significa que la variable se puede utilizar como cualquier otra variable de su tipo, pero su valor no se puede cambiar. Se recibirá un error del compilador si se intenta asignar un valor a una variable **const.**

Ejemplo

```
const float pi = 3.14;
float x;
// ....
x = pi * 2;          // Es correcto usar const's en operaciones matemáticas
pi = 7;              // esto es incorrecto - no se puede escribir (modificar) una constante
```

17. ALGUNAS FUNCIONES MATEMÁTICAS

17.1. constrain (x, a, b)

Restringe un número a estar dentro de un rango definido (limita dicho número). De los parámetros, "**x**" es la variable a limitar y "**a**" y "**b**" son los extremos inferior y superior del rango al que se limita.

Devuelve "**x**", si "**x**" está entre "**a**" y "**b**"; devuelve "**a**" si "**x**" es menor que "**a**" y "**b**" si "**x**" es mayor que "**b**".

Ejemplo

```
sensVal = constrain (sensVal, 10, 150);      // limita el rango de valores de sensVar entre 10 y 150
```

17.2. map (valor, desdeinferior, desdesuperior, hastainferior, hastasuperior)

Re-asigna un número de un rango a otro. Es decir, el valor de **desdeinferior** se asignaría a **hastainferior,** el valor de **desdesuperior** a **hastasuperior,** los valores intermedios a valores intermedios, etc. Devuelve el valor reasignado.

La función ***map()*** utiliza operaciones matemáticas de tipo entero por lo que no va a generar fracciones, aunque fuere el resultado correcto. Los restos fraccionales se truncan (no son redondeados).

Parámetros

- *valor*: el número a reasignar
- *desdeinferior*: el límite inferior del rango inicial de valores
- *desdesuperior*: el límite superior del rango inicial de valores
- *hastainferior*: el límite inferior del rango final de valores
- *hastasuperior*: el límite superior del rango final de valores

Ejemplo

```
/* Re-asignar un valor procedente de una lectura analógica (0 a 1023) a 8 bits (0 a 255) * /

void setup () {}

void loop ()
{
  int val = analogRead (0);              // analogRead devuelve un valor entre 0 y 1023
  val = map (val, 0, 1023, 0, 255);
  analogWrite (9, val);                  // el segundo parámetro de analogWrite es un valor entre 0 y 255 que
                                         // se corresponden con una tensión entre 0 y 5 V respectivamente.
}
```

17.3. pow (base, exponente)

Calcula el resultado de elevar el parámetro *base* al valor del parámetro *exponente*. Los parámetros son de tipo *float*. Devuelve el resultado en tipo *double* (en Arduino igual que el *float*).

pow() también se puede utilizar para elevar un número a una potencia fraccionaria.

Ejemplo

```
float result;
float miExponente;
result = pow(10, miExponente);
```

18. FUNCIONES CREADAS POR EL PROGRAMADOR

Las funciones permiten crear porciones modulares de código que realizan una tarea y luego vuelven a la zona de código desde la que fueron "llamadas". Son muy útiles cuando hay que realizar la misma acción varias veces en un mismo programa.

La estandarización de los fragmentos de código en funciones tiene varias ventajas:

- Las funciones codifican una acción en un solo lugar de forma que la función sólo tiene que ser diseñada y depurada una vez.
- Se reducen las posibilidades de errores cuando hay que hacer modificaciones.
- Las funciones hacen el programa más pequeño y compacto, porque sus secciones de código se reutilizan muchas veces.
- Hacen más fácil la comprensión del código y su reutilización en otros programas.

Hay dos funciones necesarias en un programa de Arduino, setup() y loop(). El resto de las funciones se deben crear fuera de las llaves de estas dos funciones. Como ejemplo, vamos a crear una función simple para multiplicar dos números.

Ejemplo 1

```
int miFuncionMultiplicadora (int x, int y) {
        int resultado;
        resultado = x * y;
        return resultado;
}
```

int ante el nombre de la función indica el tipo de datos devuelto (si no devuelve nada es “void”).

int x e *int y* son los parámetros que hay pasar a la función, que deben ser de tipo int.

resultado es el valor que devuelve la función.

Para "llamar" a nuestra función, le pasamos los parámetros del tipo de datos que la función espera:

```
void loop {
   int i = 2;
   int j = 3;
   int k;
```

```
    k = miFuncionMultiplicadora(i, j);    // k ahora contiene 6
  }
```

Nuestra función debe ser declarada fuera de cualquier otra función, pudiendo ir por encima o por debajo de la función "loop()".

El programa completo se vería así:

```
 void setup() {
   Serial.begin (9600);
}
void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = miFuncionMultiplicadora (i, j);   // k ahora contiene 6
  Serial.println (k);
  delay(500);
}

int miFuncionMultiplicadora (int x, int y) {
  int resultado;
  resultado = x * y;
  return resultado;
}
```

Ejemplo 2

Esta función lee un sensor cinco veces con analogRead() y calcula el promedio de las cinco lecturas. A continuación, escala los datos a 8 bits (0-255) y devuelve el resultado.

```
  int LeeSensor_y_Promedia() {
  int i;
  int sval = 0;
  for (i = 0; i <5; i++) {
    sval = sval + analogRead(0); // sensor en el pin analógico 0
  }
  sval = sval /5;   // media
  sval = sval /4;   // escala a 8 bits (0-255) pues analogRead devuelve un valor entre 0 y 1023
                    // también podríamos haber hecho sval=map(sval, 0, 1023, 0, 255)
  return sval;
}
```

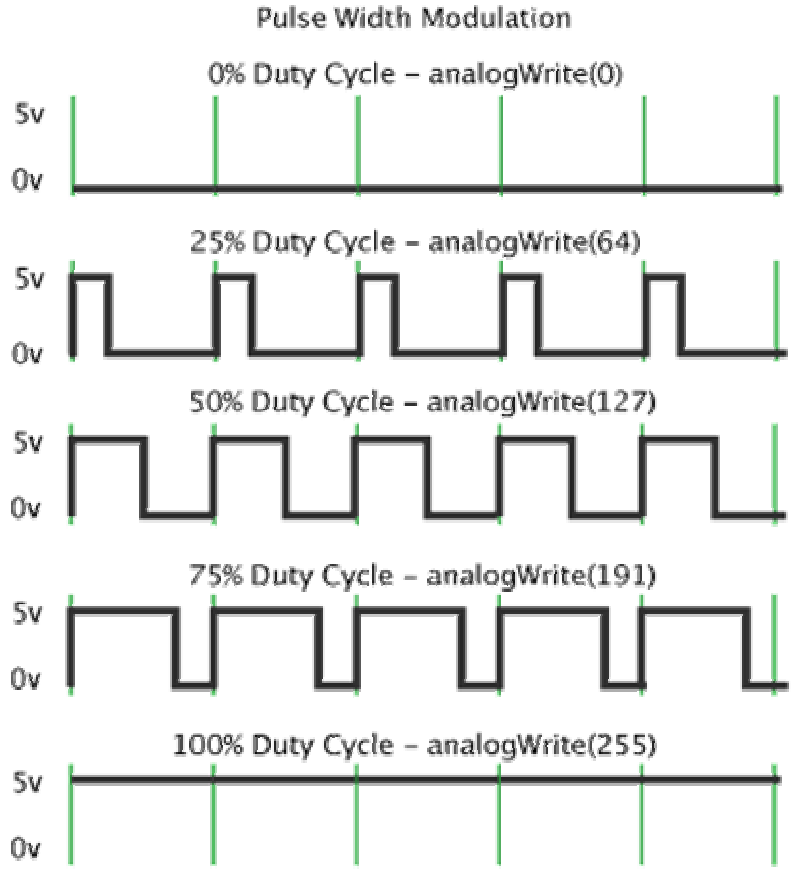
Para llamar a nuestra función tenemos que asignarla a una variable.

```
 int sens;
 sens = LeeSensor_y_Promedia ();
```

Apéndice: PWM (Modulación por anchura de pulso)

La Modulación por Ancho de Pulso (PWM = Pulse Width Modulation) es una técnica para simular un nivel de tensión analógico con una señal digital. El control digital se usa para crear una onda cuadrada, una señal que conmuta constantemente entre encendido y apagado. Este patrón de encendido-apagado puede simular voltajes entre 0 (siempre apagado) y 5 V (siempre encendido) simplemente variando la proporción de tiempo entre encendido y apagado. A la duración del tiempo de encendido (ON) se le llama *Ancho de Pulso* (pulse width). Para variar el valor analógico, cambiamos, o modulamos, ese *ancho de pulso*. Si repetimos este patrón de encendido-apagado lo suficientemente rápido, por ejemplo con un LED, el resultado es como si la señal adoptara un valor analógico entre 0 y 5 V, controlando el brillo del LED.

En el gráfico adjunto las líneas verdes representan un periodo regular que, con Arduino es de unos 2 milisegundos cada uno. La llamada a la función analogWrite() debe ser en la escala desde 0 a 255, siendo 255 el 100% de ciclo (siempre encendido), el valor 127 será el 50% del ciclo (la mitad del tiempo encendido), etc.



El ejemplo "Fading" de Arduino demuestra el uso de una salida analógica (PWM) para atenuar y aumentar la luminosidad de un LED. Está disponible en la opción "File->Sketchbook->Examples->Analog" del menú del software de Arduino.