COMENZANDO CON ARDUINO

5.1. Estructura

La estructura básica del lenguaje de programación Arduino es bastante simple y se organiza en al menos dos partes o funciones que encierran bloques de declaraciones.

```
void setup()
{
  statements;
}

void loop()
{
  statements;
}
```

Ambas funciones son requeridas para que el programa funcione.

setup()

La función *setup* debería contener la declaración de cualquier variable al comienzo del programa. Es la primera función a ejecutar en el programa, es ejecutada una vez y es usada para asignar *pinMode* o inicializar las comunicaciones serie.

```
void setup()
{
  pinMode(pin, OUTPUT); //ajusta 'pin' como salida
}
```

loop()

La función *loop* se ejecuta a continuación e incluye el código que se ejecuta contínuamente - leyendo entradas, activando salidas, etc. Esta función es el núcleo de todos los programas Arduino y hace la mayor parte del trabajo.

```
void loop()
{
  digitalWrite(pin, HIGH); //Activa 'pin'
  delay(1000);             //espera un segundo
  digitalWrite(pin, LOW);  //Desactiva 'pin'
  delay(1000);             //espera un segundo
}
```

funciones

Una función es un bloque de código que tiene un nombre y un grupo de declaraciones que se ejecutan cuando se llama a la función. Podemos hacer uso de funciones integradas como *void setup()* y *void loop()* o escribir nuevas.

Las funciones se escriben para ejecutar tareas repetitivas y reducir el desorden en un programa. En primer lugar se declara el tipo de la función, que será el valor retornado por la función (*int, void...*). A continuación del tipo, se declara el nombre de la función y, entre paréntesis, los parámetros que se pasan a la función.

```
type functionName(parameters)
{
  statements;
}
```

La siguiente función *int delayVal()*, asigna un valor de retardo en un programa por lectura del valor de un potenciómetro.

```
int delayVal()
{
  int v;               //crea una variable temporal 'v'
  v = analogRead(pot); //lee el valor del potenciómetro
  v /= 4;              //convierte 0-1023 a 0-255
  return v;            //devuelve el valor final de v
}
```

llaves {}

Las llaves definen el comienzo y el final de bloques de función y bloques de declaraciones como *void loop()* y sentencias *for* e *if*. Las llaves deben estar balanceadas (a una llave de apertura { debe seguirle una llave de cierre }). Las llaves no balanceadas provocan errores de compilación.

```
void loop()
{
  statements;
}
```

El entorno Arduino incluye una práctica característica para chequear el balance de llaves. Sólo selecciona una llave y su compañera lógica aparecerá resaltada.

punto y coma ;

Un punto y coma debe usarse al final de cada declaración y separa los elementos del programa. También se usa para separar los elementos en un bucle *for*.

```
int x = 13; //declara la variable 'x' como el entero 13
```

Nota: Olvidar un punto y coma al final de una declaración producirá un error de compilación.

bloques de comentarios /*...*/

Los bloques de comentarios, o comentarios multilínea, son areas de texto ignoradas por el programa y se usan para grandes descripciones de código o comentarios que ayudan a otras personas a entender partes del programa. Empiezan con */** y terminan con ***/** y pueden abarcar multiples lineas.

```
/*
 este es un bloque de comentario encerrado
 no olvides cerrar el comentario
 tienen que estar balanceados!
*/
```

Como los comentarios son ignorados por el programa y no ocupan espacio en memoria deben usarse generósamente y también pueden usarse para «comentar» bloques de código con propositos de depuración.

comentarios de línea //

Comentarios de una línea empiezan con *//* y terminan con la siguiente línea de código. Como el bloque de comentarios, son ignorados por el programa y no toman espacio en memoria.

```
// este es un comentario de una línea
```

Comentarios de una línea se usan a menudo despues de declaraciones válidas para proporcionar más información sobre qué lleva la declaración o proporcionar un recordatorio en el futuro.

5.2. Variables

Una variable es una forma de llamar y almacenar un valor numérico para usarse después por el programa. Como su nombre indica, las variables son números que pueden cambiarse continuamente al contrario que las constantes, cuyo valor nunca cambia. Una variable necesita ser declarada y, opcionalmente, asignada al valor que necesita para ser almacenada.

```
int inputVariable = 0;          //declara una variable y asigna el valor a 0
inputVariable = analogRead(2); //ajusta la variable al valor del pin
                                //analógico 2
```

Una vez que una variable ha sido asignada, o reasignada, puedes testear su valor para ver si cumple ciertas condiciones, o puedes usarlo directamente.

```
if(inputVariable < 100)  //comprueba si la variable es menor que 100
{
  inputVariable = 100;   //si es cierto asigna el valor 100
}
delay(inputVariable);    //usa la variable como retardo
```

declaración de variable

Todas las variables tienen que ser declaradas antes de que puedan ser usadas. Declarar una variable significa definir su tipo de valor, como *int*, *long*, *float*, etc., definir un nombre específico, y, opcionalmente, asignar un valor inicial. Esto sólo necesita hacerse una vez en un programa pero el valor puede cambiarse en cualquier momento usando aritmética y varias asignaciones.

```
int inputVariable = 0;
```

Una variable puede ser declarada en un número de posiciones en todo el programa y donde esta deficinición tiene lugar determina que partes del programa pueden usar la variable.

ámbito de la variable

Una variable puede ser declarada al comienzo del programa antes del *void setup()*, localmente dentro de funciones, y algunas veces en un bloque de declaración, por ejemplo bucles *for*. Donde la variable es declarada determina el ámbito de la variable, o la abilidad de ciertas partes de un programa de hacer uso de la variable.

Una variable global es una que puede ser vista y usada por cualquier función y declaración en un programa. Esta variable se declara al comienzo del programa, antes de la función *setup()*.

Una variable local es una que se define dentro de una función o como parte de un bucle *for*. Sólo es visible y sólo puede ser usada dentro de la función en la cual fue declarada. Además, es posible tener dos o más variables del mismo nombre en diferentes partes del programa que contienen diferentes valores.

```
int value; //'value' es visible por cualquier función

void setup()
{
  //no se necesita setup
}

void loop()
{
  for(int i=0; i<20;) //'i' es sólo visible dentro del bucle for
  {
    i++;
```

```
  }
  float f; //'f' es sólo visible dentro de loop
}
```

5.3. Tipos de datos

byte

Byte almacena un valor numérico de 8 bits sin puntos decimales. Tienen un rango de 0 a 255.

```
byte someVariable = 180; //declara 'someVariable' como un tipo byte
```

int

Enteros son los tipos de datos primarios para almacenamiento de números sin puntos decimales y almacenan un valor de 16 bits con un rango de -32,768 a 32,767.

```
int someVariable = 1500; //declara 'someVariable' como tipo int
```

long

Tipo de datos de tamaño extendido para enteros largos, sin puntos decimales, almacenados en un valor de 32 bits con un rango de -2,146,483,648 a 2,147,483,647.

```
long someVariable = 90000; //declara 'someVariable' como tipo long
```

float

Un tipo de datos para números en punto flotante, o números que tienen un punto decimal. Los números en punto flotante tienen mayor resolución que los enteros y se almacenan como valor de 32 bits con un rango de -3.4028235E+38 a 3.4028235E+38.

```
float someVariable = 3.14; //declara 'someVariable' como tipo float
```

arrays

Un array es una colección de valores que son accedidos con un índice numérico. Cualquier valor en el array debe llamarse escribiendo el nombre del array y el índice numérico del valor. Los arrays están indexados a cero, con el primer valor en el array comenzando con el índice número 0. Un array necesita ser declarado y opcionalmente asignarle valores antes de que puedan ser usados.

```
int myArray[] = {value0, value1, value2...};
```

Asimismo es posible declarar un array declarando el tipo del array y el tamaño y luego asignarle valores a una posición del índice.

```
int myArray[5];  //declara un array de enteros con 6 posiciones
myArray[3] = 10; //asigna a la cuarta posición del índice el valor 10
```

Para recibir un valor desde un array, asignamos una variable al array y la posición del índice:

```
x = myArray[3]; //x ahora es igual a 10
```

5.4. Aritmética

Los operadores aritméticos incluyen suma, resta, multiplicación y división. Retornan la suma, diferencia, producto o cociente (respectivamente) de dos operandos.

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

La operación es llevada a cabo usando del tipo de datos de los operandos, así $9/4$ devuelve 2 en lugar de 2.25. Si los operandos son de tipos diferentes, el tipo mayor es usado para el cálculo.

Nota: Usar el operador *cast*, por ejemplo *(int)myFloat* para convertir un tipo de variable a otro al vuelo.

asignaciones compuestas

Las asignaciones compuestas combinan una operación aritmética con una asignación de variable. Estas son muy frecuentemente encontradas en bucles *for*. Las asignaciones compuestas más comunes incluyen:

```
x++;    //lo mismo que x = x+1
x--;    //lo mismo que x = x-1
x += y; //lo mismo que x = x+y
x -= y; //lo mismo que x = x-y
x *= y; //lo mismo que x = x*y
x /= y; //lo mismo que x = x/y
```

operadores de comparación

Las comparaciones de una variable o constante con otra se usan a menudo en declaraciones *if* para comprobar si un condición específica es cierta.

```
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 que y
x >= y; //x es mayor o igual que y
```

operadores lógicos

Los operadores lógicos son normalmente una forma de comparar dos expresiones y devuelven TRUE o FALSE dependiendo del operador. Hay tres operadores lógicos, AND, OR y NOT, que se usan a menudo en declaraciones *if*.

```
//AND logico:
if(x>0 && x<5)  //verdadero sólo si las dos expresiones son ciertas

//OR logico:
if(x>0 || y>0)  //verdadero si al menos una expresion es cierta

//NOT logico:
if(!(x>0))      //verdadero sólo si la expresión es falsa
```

5.5. Constantes

El lenguaje Arduino tiene unos cuantos valores predefinidos que se llaman constantes. Se usan para hacer los programas más legibles. Las constantes se clasifican en grupos.

true/false

Estas son constantes Booleanas que definen niveles lógicos. FALSE se define como 0 (cero) mientras TRUE es 1 o un valor distinto de 0.

```
if(b == TRUE)
{
  doSomething;
}
```

high/low

Estas constantes definen los niveles de pin como HIGH o LOW y se usan cuando se leen o se escriben los pines digitales. HIGH esta definido como el nivel 1 lógico, ON ó 5 V, mientras que LOW es el nivel lógico 0, OFF ó 0 V.

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

input/output

Constantes usadas con la función *pinMode()* para definir el modo de un pin digital como INPUT u OUTPUT.

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

5.6. Control de flujo

if

Las sentencias *if* comprueban si cierta condición ha sido alcanzada y ejecutan todas las sentencias dentro de las llaves si la declaración es cierta. Si es falsa el programa ignora la sentencia.

```
if(someVariable ?? value)
{
  doSomething;
}
```

Nota: Cuídate de usar «=» en lugar de «==» dentro de la declaración de la sentencia *if*.

if... else

if... else permite tomar decisiones «este - o este».

```
if(inputPin == HIGH)
{
  doThingA;
}
else
{
  doThingB;
}
```

else puede preceder a otra comprobación *if*, por lo que multiples y mutuas comprobaciones exclusivas pueden ejecutarse al mismo tiempo.

```
if(inputPin < 500)
{
  doThingA;
}
else if(inputPin >= 1000)
{
  doThingB;
}
else
{
  doThingC;
}
```

for

La sentencia *for* se usa para repetir un bloque de declaraciones encerradas en llaves un número específico de veces. Un contador de incremento se usa a menudo para incrementar y terminar el bucle. Hay tres partes seperadas por punto y coma (*;*), en la cabecera del bucle.

```
for(inicializacion; condicion; expresion)
{
  doSomething;
}
```

La inicialización de una variable local, o contador de incremento, sucede primero y una sola una vez. Cada vez que pasa el bucle, la condición siguiente es comprobada. Si la condición devuelve TRUE, las declaraciones y expresiones que siguen se ejecutan y la condición se comprueba de nuevo. Cuando la condición se vuelve FALSE, el bucle termina.

```
for(int i=0; i<20; i++)     //declara i, comprueba si es menor
{                           //que 20, incrementa i en 1
  digitalWrite(13, HIGH);   //activa el pin 13
  delay(250);               //pausa por un 1/4 de segundo
  digitalWrite(13, LOW);    //desactiva el pin 13
  delay(250);               //pausa por un 1/4 de segundo
}
```

while

El bucle *while* se repetirá continuamente, e infinitamente, hasta que la expresión dentro del paréntesis se vuelva falsa. Algo debe cambiar la variable testeada, o el bucle *while* nunca saldrá. Esto podría estar en tu código, como por ejemplo una variable incrementada, o una condición externa, como un sensor de comprobación.

```
while(someVariable ?? value)
{
  doSomething;
}

while(someVariable < 200)     //comprueba si es menor que 200
{
  doSomething;                //ejecuta las sentencias encerradas
  someVariable++;             //incrementa la variable en 1
}
```

do... while

El bucle *do... while* es un bucle que trabaja de la misma forma que el bucle *while*, con la excepción de que la condición es testeada al final del bucle, por lo que el bucle *do... while* siempre se ejecutará al menos una vez.

```
do
{
  doSomething;
}while(someVariable ?? value);

do
{
  x = readSensors();     //asigna el valor de readSensors() a x
  delay(50);             //pausa de 50 milisegundos
}while(x < 100);         //repite si x es menor que 100
```

5.7. E/S digital

pinMode(pin, mode)

Se usa en *void setup()* para configurar un pin específico para que se comporte o como INPUT o como OUTPUT.

```
pintMode(pin, OUTPUT);  //ajusta 'pin' como salida
```

Los pines digitales de Arduino estan ajustados a INPUT por defecto, por lo que no necesitan ser declarados explícitamente como entradas con *pinMode()*. Los pines configurados como INPUT se dice que están e un estado de alta impedancia.

Hay también convenientes resistencias de *pull-up* de 20KOhm, integradas en el chip ATmega que pueden ser accedidas por software. A estas resistencias *pull-up* integradas se accede de la siguiente manera.

```
pinMode(pin, INPUT);       //ajusta 'pin' como entrada
digitalWrite(pin, HIGH);   //activa la resistencia de pull-up
```

Las resistencias de *pull-up* se usarían normalmente para conectar entradas como interruptores.

Los pines configurados como OUTPUT se dice que están en un estado de baja impedancia y pueden proporcionar 40 mA a otros dispositivos/circuitos.

Nota: Cortocircuitos en los pines de Arduino o corriente excesiva puden dañar o destruir el pin de salida, o dañar el chip ATmega. A menudo es una buena idea conectar un pin OUTPUT a un dispositivo externo en serie con una resistencia de 470Ohm o 1KOhm.

digitalRead(pin)

Lee el valor desde un pin digital especificado con el resultado HIGH o LOW. El pin puede ser especificado o como una variable o como una constante (0 - 13).

```
value = digitalRead(Pin);   //ajusta 'value' igual al pin de entrada
```

digitalWrite(pin, value)

Devuelve o el nivel lógico HIGH o LOW a (activa o desactiva) un pin digital especificado. El pin puede ser especificado como una variable o constante (0 - 13).

```
digitalWrite(pin, HIGH);   //ajusta 'pin' a HIGH

//Ejemplo de programa
int led = 13;   //conecta 'led' al pin 13
int pin = 7;    //conecta 'pushbutton' al pin 7
int value = 0;  //variable para almacenar el valor leido

void setup()
{
  pinMode(led, OUTPUT);   //ajusta el pin 13 como salida
  pinMode(pin, INPUT);    //ajusta el pin 7 como entrada
}

void loop()
{
  value = digitalRead(pin);   //ajusta 'value' igual al pin de entrada
  digitalWrite(led, value);   //ajusta 'led' al valor del boton
}
```

5.8. E/S analógica

analogRead(pin)

Lee el valor desde un pin analógico especificado con una resolución de 10 bits. Esta función sólo trabaja en los pines analógicos (0 - 5). Los valores enteros devueltos están en el rango de 0 a 1023.

```
value = analogRead(pin);   //ajusta 'value' igual a 'pin'
```

Nota: Los pines analógicos al contrario que los digitales, no necesitan ser declarados al principio como INPUT u OUTPUT.

analogWrite(pin, value)

Escribe un valor pseudo analógico usando *modulación por ancho de pulso* («PWM» en inglés) a un pin de salida marcado como PWM. En los Arduinos más nuevos con el chip ATmega168, esta función trabaja en los pines 3, 5, 6, 9, 10 y 11. Los Arduinos más antiguos con un ATmega8 sólo soporta los pines 9, 10 y 11. El valor puede ser especificado como una variable o constante con un valor de 0 a 255.

```
analogWrite(pin, value);   //escribe 'value' al 'pin' analogico
```

Valor	Nivel de salida
0	0 V (t)
64	0 V (3/4 de t) y 5 V (1/4 de t)
128	0 V (1/2 de t) y 5 V (1/2 de t)
192	0 V (1/4 de t) y 5 v (3/4 de t)
255	5 V (t)

Cuadro 5.1: Relación valor-salida con *analogWrite()*

El valor de salida varía de 0 a 5 V según el valor de entrada (de 0 a 255) en función del tiempo de pulso. Si t es el tiempo de pulso, la tabla 5.1 muestra la equivalencia entre el valor y la salida en función del tiempo.

Como esta es una función hardware, el pin generará una onda estática después de una llamada a *analogWrite* en segundo plano hasta la siguiente llamada a *analogWrite* (o una llamada a *digitalRead* o *digitalWrite* en el mismo pin).

```
int led = 10;                //LED con una resistencia de 220ohm en el pin 10
int pin = 0;                 //potenciometro en el pin analogico 0
int value;                   //valor para lectura

void setup(){}               //setup no es necesario

void loop()
{
  value = analogRead(pin);  //ajusta 'value' igual a 'pin'
  value /= 4;               //convierte 0-1023 a 0-255
  analogWrite(led, value);  //saca la señal PWM al led
}
```

5.9. Tiempo

delay(ms)

Pausa tu programa por la cantidad de tiempo especificada en milisegundos, donde 1000 es igual a 1 segundo.

```
delay(1000);   //espera por un segundo
```

millis()

Devuelve el número de milisegundos desde que la placa Arduino empezó a ejecutar el programa actual como un valor *long* sin signo.

```
value = millis(); //ajusta 'value' igual a millis()
```

Nota: Este número se desbordará (resetear de nuevo a cero), después de aproximadamente 9 horas.

5.10. Matemáticas

min(x,y)

Calcula el mínimo de dos números de cualquier tipo de datos y devuelve el número más pequeño.

```
value = min(value, 100);   //asigna a 'value' al más pequeño de 'value' o 100,
                           //asegurandose que nunca superara 100.
```

max(x,y)

Calcula el máximo de dos números de cualquier tipo de datos y devuelve el número más grande.

```
value = max(value, 100);    //asigna a 'value' al más grande de 'value' o 100,
                            //asegurandose que es al menos 100.
```

5.11. Aleatorio

randomSeed(seed)

Asigna un valor, o semilla («seed» en inglés), como el punto de partida para la función *random()*.

```
randomSeed(value);   //asigna 'value' como la semilla aleatoria
```

Como el Arduino es incapaz de crear un número verdaderamente aleatorio, *randomSeed* te permite colocar una variable, constante, u otra función dentro de la función *random*, lo cual ayuda a generar mas números *«random»* aleatorios. Existen una variedad de diferentes semillas, o funciones, que pueden ser usadas en esta función incluyendo *millis()* o incluso *analogRead()* para leer ruido eléctrico a través de un pin analógico.

random(max)
random(min, max)

La función *random* te permite devolver números pseudo aleatorios en un rango especificado por los valores *min* y *max*.

```
value = random(100, 200);   //asigna a 'value' un número aleatorio
                            //entre 100 y 200.
```

Nota: Usar esto después de usar la función *randomSeed()*.

```
int randNumber;                 //variable para almacenar el valor
                                //aleatorio
int led = 10;                   //LED con una resistencia de 220ohm
                                //en el pin 10
```

```
void setup(){}                    //setup no es necesario

void loop()
{
  randomSeed(millis());           //asigna millis() como semilla
  randNumber = random(255);       //numero aleatorio de 0 a 255
  analogWrite(led, randNumber);   //salida de la señal PWM
  delay(500);
}
```

5.12. Serie

Serial.begin(rate)

Abre el puerto serie y asigna la tasa de baudios para la transmisión de datos serie. La típica tasa de baudios para comunicarse con el ordenador es 9600 aunque otras velocidades están soportadas.

```
void setup()
{
  Serial.begin(9600);  //abre el puerto serie
                       //ajusta la tasa de datos a 9600 bps
}
```

Nota: Cuando se usa la comunicación serie, los pines digitales 0 (Rx) y 1 (Tx) no pueden ser usados al mismo tiempo.

Serial.println(data)

Imprime datos al puerto serie, seguido de un retorno de carro y avance de línea automáticos. Este comando toma la misma forma que *Serial.print()*, pero es más fácil para leer datos en el *Serial Monitor*[1].

```
Serial.println(analogValue);   //envia el valor de 'analogValue'

//Ejemplo de aplicacion
void setup()
{
  Serial.begin(9600);             //ajusta al serie a 9600 bps
}

void loop()
{
  Serial.println(analogRead(0));  //envia valor analogico
  delay(1000);                    //pausa por 1 segundo
}
```

[1]Más información en: *4.2 Introducción al Entorno Arduino*