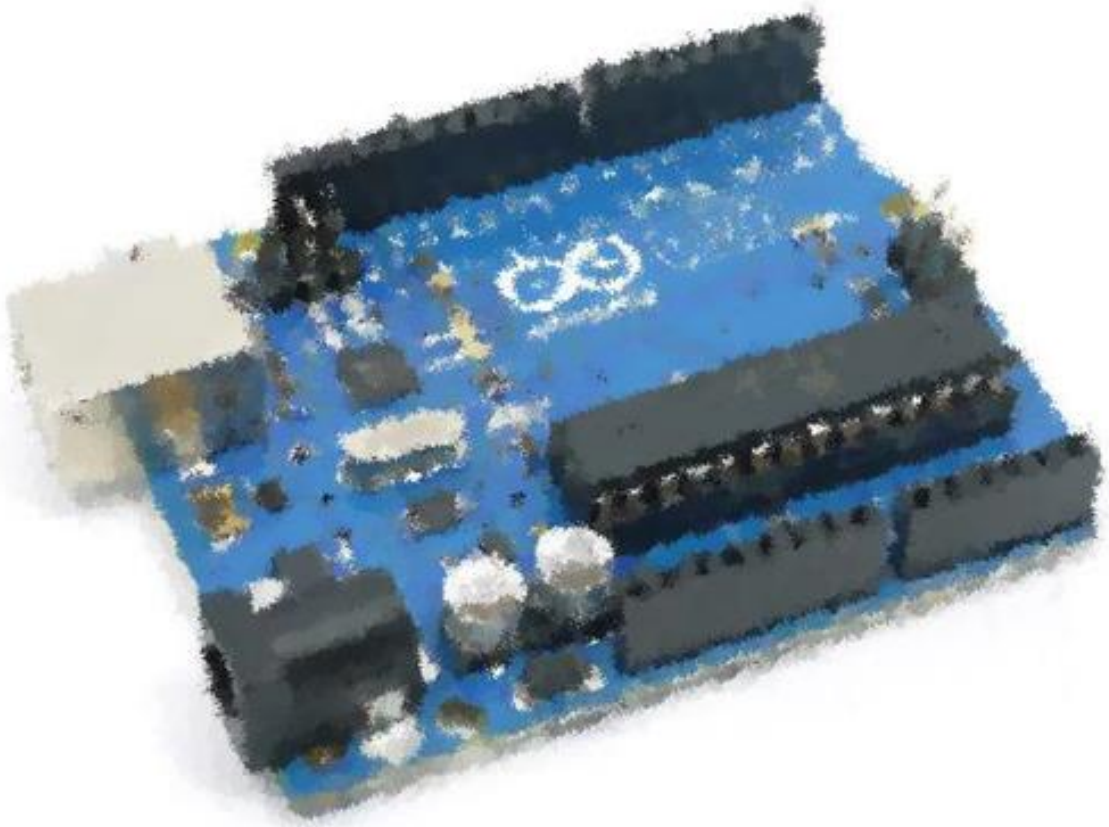
CAPÍTULO

4 nivel enteraillo

Genuino ARDUINO



Daniel Gallardo García
Profesor de Tecnología
Jerez de la Frontera

Índice

Daniel Gallardo García
Profesor de Tecnología
Jerez de la Frontera

Características avanzadas de Arduino

Microcontrolador:	ATmega328
Tensión de funcionamiento:	5 V
Tensión de entrada (recomendada):	7 – 12 V
Pines de Entradas/Salidas Digitales:	14 (6 proporcionan PWM)
Pines de Entradas Analógicas:	6
Intensidad C.C. por pines de E/S:	40 mA
Intensidad C.C. por el pin de 3,3 V:	50 mA
Intensidad C.C. por el pin de 5 V:	300 mA
Memoria Flash:	32 KB (0,5 KB para bootloader)
SRAM:	2 KB
EEPROM:	1 KB
Frecuencia señal de reloj:	16 MHz

Información sobre algunos pines

pin0 RX←: se usa para recibir datos del PC (*serial data*) o de cualquier otro dispositivo con el que queramos comunicar nuestra Arduino. En caso de establecer una comunicación Serial en nuestro programa, no debemos conectar nada a este pin.

pin1 TX→: se usa para transmitir datos al PC (*serial data*) o a cualquier otro dispositivo con el que queramos comunicar nuestra Arduino. En caso de establecer una comunicación Serial en nuestro programa, no conectaremos nada a este pin.

RESET: funciona igual que el botón de reset. Cuando a este pin le llega un pulso de tensión de valor de 0 V (es decir, poner 0 V durante un pequeño tiempo), la Arduino se resetea, comenzando a ejecutar desde el principio el programa que esté cargado en su memoria.

Información sobre algunos pines_2

Los **pines A0, …, A5** también pueden utilizarse como entradas digitales o como salidas analógicas y digitales. No tenemos más que incluirlas en el `void setup()` como los siguientes pines después del `13`, es decir: el `14, 15, 16, 17, 18, 19` (que corresponden al `A0, A1, A2, A3, A4, A5`):

```
pinMode(16, OUTPUT);  //utilizaré el pin A2 como salida
```

No es estrictamente necesario **configurar los pines** como `INPUT` u `OUTPUT` dentro de `void setup()`. Podemos cambiar durante el transcurso del sketch si un pin actuará como salida o como entrada.
Simplemente debemos interpretar al bloque `void setup()` como una parte del código que Arduino solamente la corre una vez (al principio), y en la que podemos ejecutar cualquier función (no solamente la configuración de los pines).

Información sobre algunos pines_3

AREF: en el caso de utilizar sensores que generen un rango de tensión por debajo de 5 V, podemos incrementar la precisión de la lectura haciendo que los `1024` valores posible no vayan desde los 0 V a los 5 V, sino a un rango menor de tensión (de 0 a 1,1 V; o de 0 a 3,3 V). Para ello empleamos la función `analogReference()`, que presenta tres variantes:

```
analogReference(INTERNAL);  // toma como tensión de referencia 1,1 V

analogReference(EXTERNAL);  /* toma como referencia la tensión que haya en el pin
AREF. Si quiero que esa tensión sea 3,3 V, lo único que tendré que hacer será
conectar el pin 3,3V con el pin AREF */

analogReference(DEFAULT);   // toma el valor por defecto: 5 V
```

Para que surjan efecto los nuevos valores de referencia de la tensión en las entradas analógicas, debemos llamar a la función `analogReference()` antes de utilizar la función `analogRead()`.

Tipos de variables

`boolean`: almacena un valor con dos posibilidades: 0 o 1, o verdadero o falso.

`char`: almacena un caracter, como una letra o símbolo. También se puede emplear para un número entero entre -128 a 127 (1 byte).

`byte`: almacena un número natural entre 0 y 255 (1 byte).

`int`: almacena un número entero entre -32769 y 32767 (2 bytes).

`unsigned int`: almacena un número natural entre 0 y 65536 (2 bytes).

`long`: almacena un número entero entre -2147483648 y 2147483647 (4 bytes).

`unsigned long`: almacena un número entero entre 0 y 4294967295 (4 bytes).

`float`: almacena un número decimal con un rango entre $-3.4028235 \cdot 10^{38}$ y $3.4028235 \cdot 10^{38}$ (4 bytes).

`double`: en el lenguaje C, almacenaría un número decimal con muchísima precisión, con un valor máximo de $1,7976931348623157 \cdot 10^{308}$. Sin embargo, en Arduino es lo mismo que float (4 bytes).

Tipos de variables_2

`const`: especifica que la variable definida no podrá ser cambiada durante el programa, siendo siempre un valor constante:

```
const float pi = 3.1415;
```

`#define`: Esta es otra posibilidad para poder "bautizar" a una valor constante:

```
#define pi 3.1415  //fijémonos que no acaba en ";"
```

Con esta opción, Arduino almacena el código del programa sustituyendo el nombre (en este caso `pi`) por su valor. De esta forma se consigue más velocidad ejecutando el programa. Únicamente no interesaría en el caso de que el argumento almacenado ocupe mucha memoria, y prefiramos sacrificar rapidez por memoria. Si consideramos el siguiente ejemplo:

```
#define frase "Hola, cómo están ustedes? Yo bien, gracias"
```

En ese caso conviene almacenarlo como `const char*` si se va a utilizar varias veces durante el programa, pues así solo gasta espacio en la memoria de Arduino una sola vez.

Tipos de variables_3

Una variable ya declarada se puede **cambiar de tipo de variable** durante el programa:

```
float pi = 3.1415;
int x;
x = int(pi);        /*x pasará a valer 3, pues es el resultado de
                    eliminar la parte decimal */
```

Es igualmente válida la siguiente nomenclatura:

```
x = (int)pi;
```

En el caso de que vayamos a **declarar varias variables del mismo tipo**, podemos acortar nuestro código separando las variables por coma:

```
int a = 6, b = 9, c = 11;
```

Estructuras: if y for

Ya vimos en el capítulo 2 estas dos estructuras:

Condicional:

```
if(x < 10) {
  …
}
```

```
if(x < 10) {
  …
} else {
  …
}
```

```
if(x < 10) {
  …
} else if(x < 20) {
  …
} else if(x < 30) {
  …
} else {
  …
}
```

Bucle `for`:

```
for(int i=4; i<10; i=i+1) {
  …
}
```

Veamos ahora otro tipo de estructuras: `switch`, `while` y `do-while`

Estructuras: Switch `switch (x) {case 1:…}`

Esta estructura se utiliza en el caso de que queramos que existan varios comportamientos distintos en función del valor de alguna variable:

```
switch(x) {
  case 3:          //en caso de que val==3
    …
  break;           //provoca la salida del bloque switch (opcional)
  case 12:         //también se puede poner como case(val==12):
    …
  break;           //opcional (si no quiero seguir comprobando casos)
  …
  default:         //en caso de no cumplirse ningún case, ejecutará las funciones…
    …              //…que se incluyan en el default
}
```

En el caso de que tenga que **comparar la variable con un caracter**, deberé ponerlo de la siguiente forma:

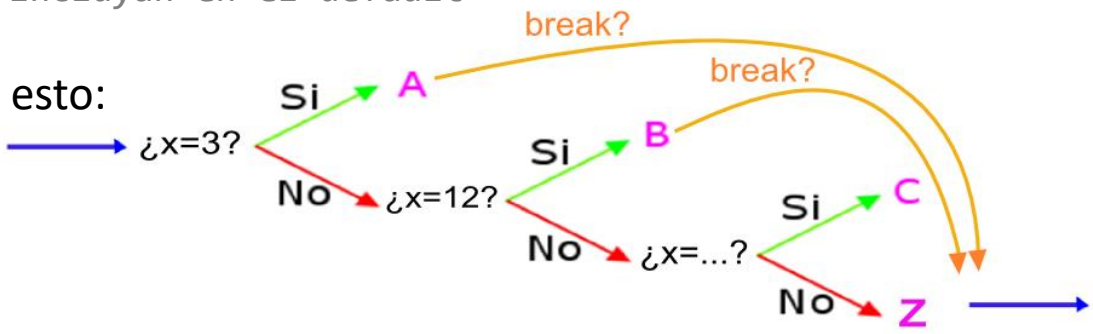
```
case(val == 'A'):    //es el caso en el que la variable es igual a la letra A
```

Estructuras: Switch `switch (x) {case 1:…}`

Esta estructura se utiliza en el caso de que queramos que existan varios comportamientos distintos en función del valor de alguna variable:

```
switch(x) {
  case 3:          //en caso de que val==3
    … A
  break;           //provoca la salida del bloque switch (opcional)
  case 12:         //también se puede poner como case(val==12):
    … B
  break;           //opcional (si no quiero seguir comprobando casos)
  …
  default:         //en caso de no cumplirse ningún case, ejecutará las funciones…
    … Z            //…que se incluyan en el default
}
```

Gráficamente sería algo como esto:



En el caso de que tenga que **comparar la variable con un caracter**, deberé ponerlo de la siguiente forma:

```
case(val == 'A'):    //es el caso en el que la variable es igual a la letra A
```

Estructuras: Bucle While

`while(x < 3) {…}`

Es otra forma de crear un bucle en Arduino. De hecho, `loop()` es de por sí un bucle infinito; pero dentro de ese bucle infinito puedo insertar otros bucles. La estructura `while` realizará el bucle mientras se cumpla cierta condición.

```
while(x < 50) {
  //mientras x sea menor de 50, ejecutará este bloque de código
}
  /*cuando no se cumpla la condición, continuará ejecutándose el
  resto del  programa */
```

Estructuras: Bucle While

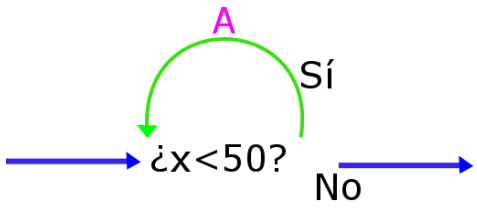
`while(x < 3) {…}`

Es otra forma de crear un bucle en Arduino. De hecho, `loop()` es de por sí un bucle infinito; pero dentro de ese bucle infinito puedo insertar otros bucles. La estructura `while` realizará el bucle mientras se cumpla cierta condición.

```
while(x < 50) {
  //mientras x sea menor de 50, ejecutará este bloque de código
}
  /*cuando no se cumpla la condición, continuará ejecutándose el
  resto del  programa */
```

A

Gráficamente sería algo como esto:



Estructuras: Bucle Do While `do {…} while(x < 3)`

Básicamente es similar a la estructura `while`, con la diferencia de que ejecuta al menos una vez el bloque de código que está entre llaves, y luego lo seguirá haciendo mientras se cumpla la condición especificada en `while`.

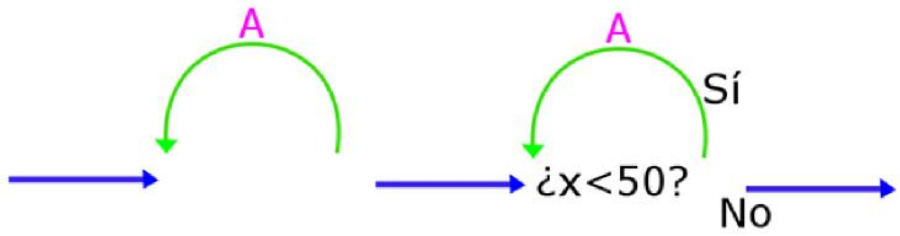
```
do {
  //al menos una vez realiza este bloque de código
} while(x < 50)
  /* si la condición se cumple, volverá a realizar el bucle, pero
  cuando no se cumpla la condición saldrá del bucle y continuará
  ejecutándose el resto del  programa */
```

Estructuras: Bucle Do While `do {…} while(x < 3)`

Básicamente es similar a la estructura `while`, con la diferencia de que ejecuta al menos una vez el bloque de código que está entre llaves, y luego lo seguirá haciendo mientras se cumpla la condición especificada en `while`.

```
do {  A
  //al menos una vez realiza este bloque de código
} while(x < 50)
  /* si la condición se cumple, volverá a realizar el bucle, pero
  cuando no se cumpla la condición saldrá del bucle y continuará
  ejecutándose el resto del  programa */
```

Gráficamente sería algo como esto:



Estructuras: break y continue `break;` `continue;`

Existen dos órdenes que pueden incluirse dentro de cualquier bucle (instrucciones comprendidas entre las llaves) de las estructuras `for`, `while` y `do while`: **`break;`** y **`continue;`**.

```
break;              /*rompe el bucle y el programa continúa por los códigos
                    que aparecen detrás del bloque */

continue;           /*salta el resto de código del bloque para comenzarlo
                    nuevamente desde el principio */
```

Veamos un ejemplo: si tenemos conectados 5 LEDs en los pines del 9 al 13, y quiero que parpadeen en secuencia solamente los que ocupan posición impar, puedo utilizar el siguiente trozo de código:

```
    for(int i=9; i<=13; i++) {
      if(i%2 == 0) continue;          //como i es par, me lo salto
      digitalWrite(i, HIGH); delay(500);
      digitalWrite(i, LOW); delay(500);
    }
/* En caso de ser i un número par, al dividirlo entre 2 dará un módulo igual a
cero, y el comando continue hará que Arduino se salte todo el resto del código
dentro del bucle for para comenzarlo de nuevo, pero incrementando i en 1. */
```

Arrays

Un array almacena una **lista de variables**, accediendo a cada una de ellas a través de su **índice** (el primero siempre es el cero):

```
int datos[] = {4, 7, 9, 12, 18};
```

Siendo: `datos[0] = 4;` `datos[1] = 7;` … `datos[4] = 18;`

En este caso no ha sido necesario especificar el tamaño del array porque hemos incluido el valor de todas las variables, pero si no declaro los valores iniciales, debo especificar el tamaño del array:

```
int datos[5];
```

Existe también otra forma de llamar los distintos elementos del array, a través de un **puntero**. El primer elemento corresponde a `*datos`, y para llamar a los siguientes no tenemos mas que sumar su posición relativa:

```
*datos     == datos[0]
*datos + 1 == datos[1]
*datos + 2 == datos[2]
       …   ==   …
*datos + n == datos[n]
```

Arrays_2: arrays multidimensionales

También es posible almacenar datos ordenados en forma de **matriz** o **array multidimensional** donde cada dato estará localizado por un doble índice (o triple, o cuádruple, etc…). En los arrays multidimensionales sí **es necesario especificar el tamaño de las respectivas dimensiones** (excepto la primera) aunque declares todos sus elementos.

```
int matriz[3][3] = { {2, 4, 8}, {3, 9, 27}, {5, 25, 125} };
```

Siendo: `matriz[0][0] = 2;` `matriz[0][1] = 4; …` `matriz[2][2] = 125;`

Arrays_3: arrays de caracteres o string

Un string (o array de caracteres) almacena una cadena de caracteres, y se emplea para almacenar textos. Existen varias opciones:

```
char letra = 'a';           //almacena un carácter (entre comillas simples)

char texto = "Me gusta Arduino!";     //almacena un texto (comillas dobles)
```

Los 18 caracteres que componen la cadena de texto `"Me gusta Arduino!"` (17 caracteres + 1 para indicar el fin de la cadena, que aunque no aparezca, está ahí: `'\0'`) son almacenados como elementos separados en el array. Por ejemplo:

```
char texto[0] = 'M';   char texto[2] = ' ';   char texto[16] = '!';
```

```
char texto[] = {'M', 'e', ' ', 'g', 'u', 's', 't', 'a', ' ', 'A', 'r', 'd',
                'u', 'i', 'n', 'o', '!'};
```

De esta otra forma, válida pero más laboriosa, declaro individualmente todos los caracteres que componen el array.

Arrays_4: arrays de caracteres o string

```
char* palabra = "Arduino";

char* frase = "Me gusta Arduino!";

char* color[] = {"Rojo", "Azul", "Verde limón"};
```

Esta otra variante, con el asterisco (*), se utiliza para almacenar palabras, frases, o un array de palabras y frases, donde cada elemento del array es un grupo de caracteres:

```
char* color[0] = "Rojo";          char* color[2] = "Verde limón";
```

Arrays_5: Tamaño de un array `sizeof(miArray)`

La función **`sizeof()`** permite reconocer el tamaño del array, en número de bytes. Veamos algunos ejemplos:

```
int edades[] = {36, 5, 68, 15, 22, 41};

char texto[] = "Me llamo Daniel";

int x = sizeof(edades);      /* x tomará un valor de 12 (cada variable
                                definida como int ocupa 2 bytes) */

int y = sizeof(texto);       /* y tomará un valor de 16 (cada carácter ocupa
                                1 bytes + 1 que indica el final de la cadena */
```

Esta función solo se utiliza en los casos en los que se emplee el tamaño del array y sea probable cambiar dicho array sin tener que cambiar el resto de código del sketch.

Para obtener el número de elementos que tiene un array, podemos utilizar:

```
int a = sizeof(edades) / 2;          /* en este ejemplo, a=6. También podría haber
                                        utilizado: sizeof(edades)/sizeof(int) */

int b = sizeof(texto) - 1;           /* en este ejemplo b=15 */
```

Ejemplo: Letra a letra

Imprimiremos un mensaje en el Monitor Serie letra a letra:

```
char mensaje[] = "Léeme despacito, pisha";     //ocupa 22+1 bytes

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

void loop() {
  for(int i=0; i<sizeof(mensaje)-1; i++) {     //imprimiré los 22 caracteres
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.println(mensaje[i], BYTE);
    delay(500);
  }
}
```

Si por ejemplo quisiera hacer algo parecido con un array del tipo `int`, debería haber puesto:

```
  for(int i=0; i<sizeof(datos)/2; i++) {         //cada número ocupa 2 bytes
```

Definir Funciones `void funcion() {… }`

Según las características de un programa, se puede volver muy útil la utilización de funciones definidas por nosotros mismos. Todo lenguaje de programación debe prestarse a esta posibilidad, y Arduino no es una excepción. Existen varias **formas de definir una función:**

`void`: Si de la función que queremos definir no se espera ningún valor de retorno, y se limita a realizar una serie de ordenes que dependan (o no) de ciertas variables, entonces la función (que seria una mera **subrutina**) se define de la siguiente manera:

```
void funcion(a, b) { … }
```

donde a y b serían variables que la función debería utilizar (por supuesto, habrá funciones que no necesiten de ninguna variable). Si dichas variables no estaban declaradas con anterioridad, habrá que declararlas en ese mismo momento:

```
void funcion(int a, int b) { … }
```

Definir Funciones_2 `int funcion() {… }`

`int`, `long`, `float`, etc: Si quiero que la función que vamos a definir **me devuelva un valor**, debo declararla como si de una variable se tratara, especificando que tipo de variable es dicho valor (`int`, `long`, `float`,…). Para que la función tome un valor, debemos utilizar la función **`return`** acompañada de la variable cuyo valor quiero asignar a la función.

```
int funcion (a, b) {          //también puede manejar variables
  int val = 0;
  … ;
  val = … ;
  return val;
}
```

Si deseo **interrumpir la función y salir de ella** (de manera análoga al `break` en las estructuras), y seguir el programa por la siguiente línea después de la llamada de la función, utilizaré el comando **`return`**; (sin acompañarlo de ninguna variable o dato, con lo cual devuelve un valor vacío).

Las funciones se suelen definir en la última parte del código, debajo del bloque de `void loop`.

Ejemplo: S.O.S.

Emplearemos el LED integrado de la placa Arduino (conectado al pin 13) para enviar una señal de socorro en código Morse: S.O.S. (Save Our Souls), es decir:

• • • (S) — — — (O) • • • (S).

Tendríamos dos opciones a la hora de realizar el programa: Hacerlo verdaderamente sencillo, con mucho *copiar* y *pegar* líneas como:

`digitalWrite(ledPin,HIGH); delay(200); digitalWrite(ledPin,LOW); delay(200);`

para cada punto o raya (la raya sería con un `delay(500)`). O acortar mucho (y por que no, embellecer) el código de nuestro programa empleando dos cosas: definiendo una función que hará el pulso de luz (flash) y un array para la duración de dichos pulsos:

```
int tiempo[] = {200, 200, 200, 500, 500, 500, 200, 200, 200};

void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  for(int i=0; i<9; i++) flash(tiempo[i]);     //“flash” es una función que…
  delay(800);                                  //…utiliza un argumento
}

void flash(int duracion) {
  digitalWrite(13, HIGH); delay(duracion);
  digitalWrite(13, LOW); delay(duracion);
}
```

Funciones de Tiempo `delay(); millis();`

Ya conocemos la función `delay();`

```
delay(1000);        //realiza una pausa en el programa de 1000 ms
```

Veamos ahora otras funciones de tiempo:

```
delayMicroseconds(350);     //realiza una pausa en el programa de 350 µs

x = millis();               /*asigna a x el número de milisegundos que han
pasado desde que la placa Arduino fue reseteada (es una variable del tipo
unsigned long). Se produciría un desbordamiento a los 50 días */

x = micros();               /*asigna a x el número de microsegundos que han
pasado desde que la placa Arduino fue reseteada (es una variable del tipo
unsigned long). Se produciría un desbordamiento a los 70 minutos */
```

Funciones de Tiempo_2

Debemos tener presente **algunas consideraciones** sobre estas funciones de tiempo:

Puede que sea necesario declarar los argumentos para `delay()` y `delayMicroseconds()` como variables del tipo `long` o `unsigned long` si lo que queremos es emplear pausas largas (del orden del minuto, por ejemplo). Si consideramos `delay(60*1000);` no hará una pausa de 1 minuto, pues al utilizar los números `60` y `1000`, Arduino los interpreta como enteros (`int`) y el resultado sobrepasa el límite de `32767`, y por el contrario daría un valor de `-5536` (`se ha rebosado`). Es posible indicar a Arduino que interprete un número como `long`, como `unsigned` o como ambos colocando tras él las letras **`l`** o **`L`**, y/o **`u`** o **`U`**:

`255u`	considera a la constante 255 como una variable del tipo `unsigned int`
`1000L`	considera a la constante 1000 como una variable del tipo `long`
`32767ul`	considera a la constante 32767 como una variable del tipo `unsiged long`

De esta forma:

```
delay(60*1000L);   //hará una pausa de 1 minuto
```

Funciones de Tiempo_3

En el caso de no querer detener el programa, por ejemplo al hacer parpadear un LED (como ocurre con `delay()`), y que pueda realizar otras tareas mientras parpadea, debemos hacer uso de la función `millis()`. Un ejemplo sería:

```
int estado = LOW;
unsigned long tiempo = 0;
unsigned long intervalo = 500;

void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  if(tiempo + intervalo < millis()) {//es la condición para que…
    estado = !estado;                          //… parpadee el LED, …
    digitalWrite(13, estado);
    tiempo = millis();
  }
  …       //… permitiendo seguir ejecutando el resto del código
}
```

Funciones de Tiempo_4 `pulseIn();`

Otra función que se utiliza en las entradas digitales es: `pulseIn(pin,val);` que determina el tiempo transcurrido hasta que se vuelva a repetir el mismo valor de lectura en la entrada digital nº pin. Es una función muy utilizada, por ejemplo, para leer sensores de infrarrojo (IR), de ultrasonido (donde Tigger y Echo sean el mismo pin) o acelerómetros, que producen como salida pulsos de duración variable.

```
x = pulseIn(7, HIGH);        /*asigna a x el tiempo transcurrido hasta el
                             próximo pulso en estado HIGH */
```

Funciones Matemáticas

`x = abs(a);`

```
x = min(a, b);      //asigna a x el valor más pequeño entre a y  b

x = max(a, b);      //asigna a x el valor más grande entre a y  b

x = abs(a);         //asigna a x el valor absoluto de  a

x = constrain(val, a, b);   /*asigna a x el valor de val siempre y cuando val esté
comprendido entre a y b. En caso de val salga de dicho intervalo, tomará los
valores extremos de éste */

x = sq(a);          //asigna a x el valor  a²

x = pow(a, b);      //asigna a x el valor  aᵇ

x = sqrt(a);        //asigna a x el valor raíz cuadrada de  a

x = sin(a);         //asigna a x el seno(a), estando a en radianes

x = cos(a);         //asigna a x el coseno(a), estando a en radianes

x = tan(a);         //asigna a x la tangente(a), estando a en radianes
```

Operadores Binarios

`x = 1 & 0;`

Arduino posee una serie de funciones con las que **operar dos bits (bit a bit)**, que permiten resolver muchos problemas comunes en programación.

& **Operador AND:** multiplicación booleana de dos bits:

```
x = bit1 & bit2;
```

&	0	1
0	0	0
1	0	1

Si por ejemplo aplicamos este operador a dos variables tipo `int`, que recordemos que eran número de 16 bits, los bits se multiplicarán uno a uno, sin tener en cuenta los acarreos. Ejemplo:

```
int a = 92;        //en binario es    0000000001011100
int b = 101;       //en binario es    0000000001100101
int x = a & b;     //el resultado es  0000000001000100, que corresponde al 68
```

Operadores Binarios_2

| ^ ~

| **Operador OR:** sumador booleano de dos bits:

```
x = bit1 | bit2;
```

|	0	1
0	0	1
1	1	1

Si por ejemplo aplicamos este operador a los dos `int` del ejemplo anterior:

```
int a = 92;       //en binario es    0000000001011100
int b = 101;      //en binario es    0000000001100101
int x = a | b;    //el resultado es  0000000001111101, que corresponde al 125
```

^ **Operador XOR:** sumador exclusivo booleano de dos bits:

```
x = bit1 ^ bit2;
```

^	0	1
0	0	1
1	1	0

~ **Operador NOT:** inversor booleano :

```
x = ~ bit1;
```

~	
0	1
1	0

Si por ejemplo aplicamos este operador a una variable tipo `int`:

```
int a = 103;      //en binario es    0000000001100111
int x = ~ a;      //el resultado es  1111111110011000, que corresponde al -104
```

Operadores Binarios_3

`x = a << 3;`

Arduino también posee dos funciones de **desplazamiento de bits**:

<< **Bitshift left:** desplaza los bits que componen una variable hacia la izquierda, rellenando los huecos que se van generando a la derecha con ceros, y perdiéndose todos aquellos bits que han “rebosado” por la izquierda.

Veamos algún ejemplo :

```
int a = 5;          //en binario es    0000000000000101
int b = a << 3;     //en binario es    0000000000101000, que corresponde al 40
int c = a << 14;    //en binario es    0100000000000000, y se pierde el primer 1
```

Es muy útil como multiplicador de potencias de 2:

```
1 << 0 == 1        1 << 1 == 2        1 << 2 == 4        1 << 3 == 8
1 << 4 == 16       1 << 5 == 32       1 << 6 == 64       1 << 7 == 128 ...
```

Operadores Binarios_4

`x = a >> 3;`

>> **Bitshift right:** desplaza los bits que componen una variable hacia la derecha, rellenando los huecos que se van generando a la derecha con ceros o unos dependiendo de si el signo del valor decimal de la variable es positivo o negativo, respectivamente, y perdiéndose todos aquellos bits que han "rebosado" por la derecha.

Veamos algún ejemplo :

```
int a = -16;                  //en binario es    1111111111110000
int b = a >> 3;               //en binario es    1111111111111110, se rellena con 1s
int c = (unsigned int)a >> 3;         //         0001111111111110, se rellena con 0s
```

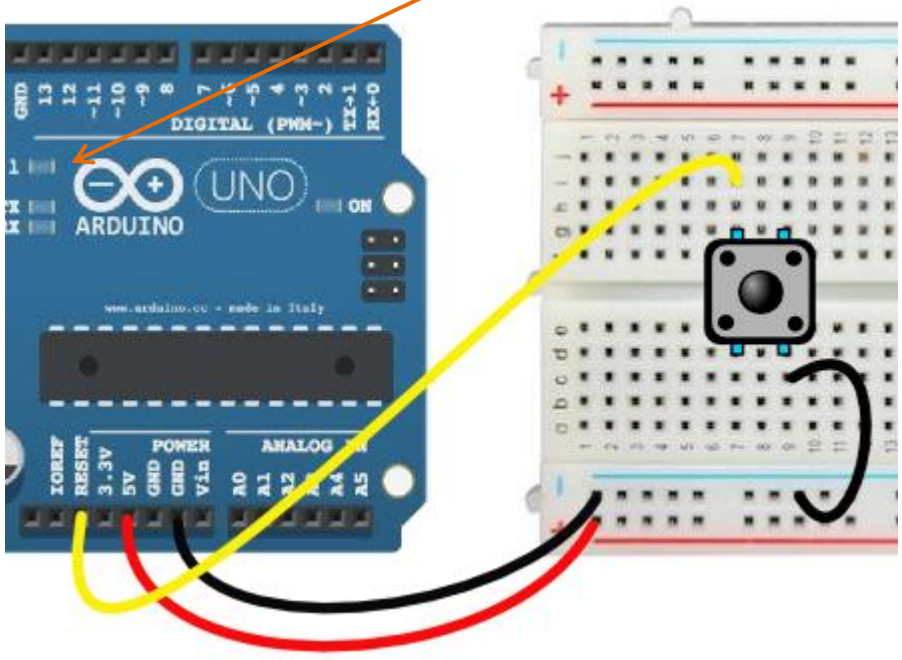
Es muy útil como divisor de potencias de 2:

```
int a = 1000;
int x = a >> 3;               //es como dividir entre 8, siendo x=125
```

Resetear Arduino

En algunas ocasiones podemos querer que nuestra Arduino se **resetee**, y que **todos los parámetros vuelvan a la situación inicial** (por ejemplo, `millis();` volvería a `0`). Es como si se volviera a bajar el programa desde el PC. Existen varias opciones:

Mediante Hardware: utilizando el pin Reset de Arduino y un pulsador. Debemos generar un pulso de 0 V en dicho pin, lo que provoca un nuevo bootloader o arranque (y se enciende el LED L integrado en Arduino). Una posible solución es la siguiente:



Cuando apriete el pulsador, Arduino se reseteará.

Resetear Arduino_2

Mediante Hardware & Software: utiliza el pin Reset de Arduino, pero ahora será una **salida digital** la que **comunique** (mediante un cable) un **pulso de 0 V** al pin Reset. La única precaución es hacer que el sketch comience poniendo a dicha salida digital un valor de `HIGH` (para que Arduino no caiga en un "coma", pues se estará reseteando eternamente, ya que los valores de salida por defecto son de `LOW`); para ello hay que escribir `digitalWrite(pin, HIGH);` en la primera línea de código del `setup()` (incluso antes que `pinMode(pin, OUTPUT);`):


```
void setup() {
  digitalWrite(2, HIGH); //evita que se resetee
  pinMode(2, OUTPUT);
  …
}

void loop() {
  …
  digitalWrite(2, LOW); //se produce el reset
  …
}
```

Resetear Arduino_3 `asm volatile("jmp 0");`

Mediante Software: Debemos **crear una función para resetar** Arduino. Las siguientes dos opciones son igualmente válidas, y el efecto es mejor que utilizando el pin Reset, pues ahora no interrumpe la corriente de Arduino y el reseteo es instantáneo (no hay que esperar ese segundo que tardaba el bootloader).

Solución 1:

```
void reseteando() {
  asm volatile("jmp 0");
}
/*puede escribirse al final
 del sketch */
```

Solución 2:

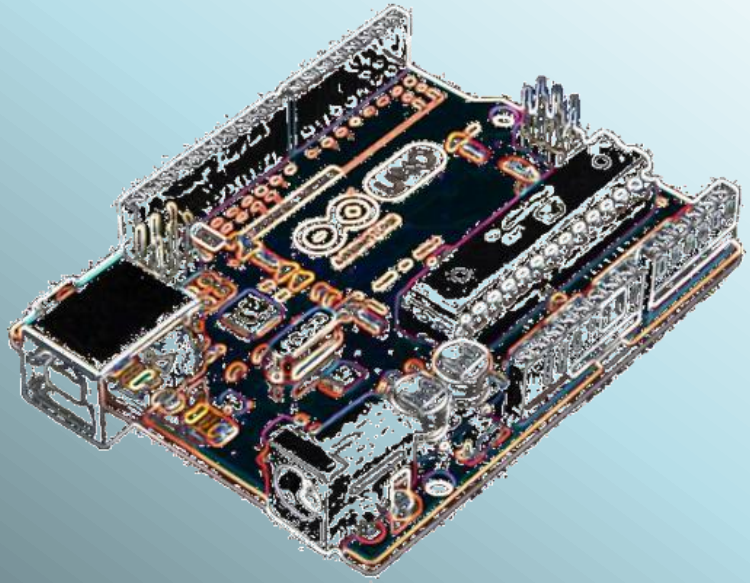
```
void(* reseteando) (void) = 0;

/*se debe escribir al principio
del sketch (antes del setup) */
```

```
/*En ambos casos, Arduino se reseteará a través del
código cuando ejecute la función: */

reseteando();
```

FIN del Capítulo IV



Daniel Gallardo García
Profesor de Tecnología
Jerez de la Frontera
danielprofedetecno@gmail.com