# GUIA A LA PROGRAMACION DE ARDUINO

Funciones



## FUNCIONES

Vimos al final de la primera parte (capítulo 5 “Funciones”) en que consistían las funciones en cualquier lenguaje de programación y en concreto en el Arduino.

Recordamos algunos de los conceptos básicos sobre funciones:

Lo primero que hay que hacer es declarar el tipo de la función (es decir, el tipo de la variable que la función va a devolver con la instrucción “*return*” (void si no devuelve ningún valor, o cualquiera de los demás tipos en caso contrario)

Si estás escribiendo una función dale un nombre significativo (es decir, que te recuerde claramente que hace esa función, que valor devuelve)

Los datos que se transfieren desde el programa a la función se llaman parámetros, pueden ser uno, varios o ninguno, y de diferentes tipos (con la excepción de las funciones como *setup()* y *loop()* que no utilizan parámetros).

El lenguaje del Arduino incluye una serie de funciones que están incluidas directamente en la librería básica (“Core”) y que podemos usar sin más. Estas funciones se pueden agrupar en diferentes tipos:

- *Funciones de entrada/salida digital (I/O: Input/Output).*
- *Funciones de entrada/salida analógicas.*
- *Funciones avanzadas de entrada/salida.*
- *Funciones de entrada/salida del Arduino Due.*
- *Funciones temporales.*
- *Funciones matemáticas.*
- *Funciones trigonométricas.*
- *Funciones aleatorias.*
- *Funciones de manipulación de bits y bytes.*
- *Funciones de gestión de interrupciones.*
- *Funciones de gestión de comunicaciones.*
- *Funciones de gestión del puerto USB (Leonardo y Due solamente).*
- *Funciones de conversión de tipos.*
- *Otras funciones.*

## FUNCIONES DE ENTRADA/SALIDA DIGITAL

## La función pinMode()

**Descripción**:

Esta función configura el pin especificado para actuar como entrada o salida digital (ver la descripción de los pins digitales del Arduino para más información sobre la funcionalidad de estos pins). A partir de la version 1.0.1 del Arduino es posible activar las resistencias internas “pullup” del Arduino con el modo INPUT_PULLUP. Por otra parte, el modo INPUT desactiva explícitamente las resistencias internas “pullup”.

**Sintaxis**:

```
pinMode(pin, modo)
```

**Parámetros**:

*pin*: el número de pin que queremos activar como entrada o salida.

*modo*: INPUT, OUTPUT, or INPUT_PULLUP (ver la descripción de los pins digitales del Arduino para más información sobre la funcionalidad de estos pins).

**Devuelve**:

Nada (esta función simplemente activa un pin sin devolver ningún valor de retorno)

**Ejemplo**:

```
int ledPin = 13;                  // LED conectado al pin digital 13
void setup()
   {
       pinMode(ledPin, OUTPUT);      // configura el pin como salida
   }

void loop()
  {
      digitalWrite(ledPin, HIGH);   // activa el LED (encendido)
      delay(1000);                  //temporizador (espera un segundo)
      digitalWrite(ledPin, LOW);    // desactiva el LED (apagado)
      delay(1000);                  // temporizador (espera un segundo)
  }
```

**Nota**:

- Los pins de entrada analógica pueden ser también usados como pins digitales si nos referimos a ellos como A0, A1, etc.

**Ver también**:

```
- constantes
- digitalWrite()
- digitalRead()
- Tutorial: Descripción de los pins del Arduino.
```

## La función digitalWrite()

**Descripción**:

Activa (HIGH) o desactiva (LOW) un pin digital.

Si el pin ha sido configurado como OUTPUT (salida) con la función *pinMode()*, su voltaje será activado a 5V (o 3.3V en las tarjetas que funcionen a 3.3V) si se activa (HIGH) o a 0V (tierra) si se desactiva (LOW).

Si el pin ha sido configurado como INPUT (entrada), *digitalWrite()* activará (si la usamos el parámetro HIGH) o desactivará (con LOW) la resistencia “pullup” del pin de entrada especificado. Se recomienda activar la resistencia interna “pullup” del pin con la función *pinMode(pin, INPUT_PULLUP)*. Ver la descripción de los pins digitales del Arduino para más información sobre la funcionalidad de estos pins).

Nota:

Si no configuras el pin como salida (OUTPUT) con la función pinMode() y conectas el pin a un LED, cuando uses la función *digitalWrite(HIGH)*, el LED no se encenderá. En efecto, si no configuras explícitamente el pin como salida con *pinMode(), digitalWrite()* activa la resistencia interna "pullup" que actúa como un potente limitador de corriente.

**Sintáxis**:

```
digitalWrite(pin, valor)
```

**Parámetros**:

*pin*: el número de pin

*valor*: HIGH or LOW

**Devuelve**:

nada

**Ejemplo**:

```
int ledPin = 13;                // LED conectado al pin digital 13
void setup()
   {
      pinMode(ledPin, OUTPUT);      // configura el pin digital pin como salida
   }

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

Este sketch active el pin 13 (HIGH) espera un Segundo y lo desactiva.

**Nota**:

Los pins de entrada analógica pueden ser también usados como pins digitales si nos referimos a ellos como A0, A1, etc.

**Ver también**:

```
-pinMode()
-digitalRead()
-Tutorial: Digital Pins
```

## La function digitalRead()

**Descripción**:

Lee el valor (HIGH o LOW) del pin digital especificado.

**Sintaxis**:

```
digitalRead(pin)
```

**Parámetros**:

*pin:* el número de pin digital que quieres leer (tipo *int*)

**Devuelve**:

HIGH (alta) o LOW (baja/tierra)

**Ejemplo**:

Activa el pin 13 al mismo valor que el leído en el pin 7 (configurado como entrada)

```
int ledPin = 13; // LED conectado al pin digital13
int inPin = 7;   // pulsador conectado al pin digital 7
int val = 0;     // variable que almacena el valor leído

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

void loop()
  {
       val = digitalRead(inPin);   // lee valor en pin de entrada
       digitalWrite(ledPin, val);    // active el LED con el valor leído en el pulsador
  }
```

**Nota**:

Si el pin no está conectado a nada, *digitalRead()* puede devolver indistintamente HIGH o LOW (y esto puedo cambiar de manera aleatoria).

Los pins de entrada analógica pueden ser también usados como pins digitales si nos referimos a ellos como A0, A1, etc.

**Ver también**:

```
-pinMode()
-digitalWrite()
-Tutorial: Digital Pins
```

## FUNCIONES DE ENTRADA/SALIDA ANALÓGICAS

### La función analogReference()

**Descripción**:

Esta función configura el voltaje de referencia usado como entrada analógica (es decir, el valor que queremos definir como voltaje máximo de entrada). Las siguientes opciones son válidas:

-DEFAULT: el valor de referencia por defecto es 5 voltios (en las tarjetas Arduino que funcionan a 5V) o 3.3 voltios (en las Arduino que van a 3.3V)

-INTERNAL: un valor de referencia interno fijado en 1.1 volts en el ATmega168 o ATmega328 y  2.56 volts en el ATmega8 (no disponible en el Arduino Mega)

-INTERNAL1V1: un valor de referencia interno fijado en 1.1V (Solamente en el Arduino Mega)

-INTERNAL2V56: un valor de referencia interno fijado en 2.56V (Solamente en el Arduino Mega)

-EXTERNAL: el voltaje aplicado al pin AREF pin (de 0 a 5V sólamente) es usado como referencia máxima.

**Sintáxis**:

```
analogReference(tipo)
```

**Parámetros**:

*tipo*: El tipo de referencia usado (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, or EXTERNAL).

**Devuelve**:

Nada.

**Nota**:

Las lecturas efectuadas con *analogRead()* inmediatamente tras cambiar la referencia analógica pueden no ser fiables.

**Advertencia**:

¡No uses menos de 0V o más de 5V como voltaje de referencia externa en el pin AREF! SI estás usando una referencia externa en el pin AREF debes configurar la referencia como EXTERNAL antes de usar la función analogRead(). Si no lo haces así cortocircuitarás el voltaje de referencia activo (generado internamente) y el pin AREF y, muy probablemente dañarás el micro controlador del Arduino.

Por otra parte, puedes conectar el voltaje de referencia externa al pin AREF a través de una resistencia de 5K, lo cual te permitirá pasar de voltajes de referencia internos a externos. Advierte que la resistencia cambiará el voltaje que es usado como referencia ya que el pin AREF está conectado internamente a una resistencia de 32K. La combinación de estas dos resistencias (interna y externa) actúa como un divisor de voltaje. Por ejemplo, 2.5V aplicados a través de la resistencia darán 2.5 * 32 / (32 + 5) = ~2.2V en el pin AREF.

**Ver también**:

> - Descripción de los pins de entrada analógica.
> - *analogRead()*

## La función analogRead()

### Description:

Esta función lee el valor presente en el pin analógico especificado. El Arduino lleva un convertidor analógico/digital de 10 bits con 6 canales (8 canales en el Mini y Nano y 16 en el Mega). Esto significa que se pueden convertir voltajes que varíen entre 0 y 5 voltios en valores enteros entre 0 y 1023. Esto equivale a una resolución de 5V/1024 unidades, o lo que es lo mismo 0.0049 Volts (4.9 mV) por unidad. El rango de entrada y la resolución pueden ser cambiados usando *analogReference()*.

La frecuencia máxima de lectura es de 10,000 veces por segundo (cada lectura dura unos 100 microsegundos).

**Sintaxis**:

```
analogRead(pin)
```

**Parámetros**:

*pin*: el número de pin analógico del que se quiere leer la entrada (0 a 5 en la mayor parte de las placas, 0 a 7 en el Mini y Nano, 0 a15 en el Mega)

**Devuelve**:

*int* (de 0 a 1023)

**Nota**:

Si el pin de entrada analógico no está conectado a nada, el valor devuelto por la función *analogRead()* fluctuará dependiendo de una serie de factores (por ejemplo, los valores de otras entradas analógicas, la proximidad de tu mano a la placa, etc.)

**Ejemplo**:

```
int analogPin = 3;      // potenciómetro conectado al pin analógico 3
                        // variando entre tierra y +5V
int val = 0;            // variable de almacenaje del valor leído

void setup()
  {
```

```
    Serial.begin(9600);            // inicialización puerto serie a 9600 bps
  }

void loop()
  {
     val = analogRead(analogPin);   //leer entrada analógica
     Serial.println(val);           // mostrar valor leido
  }
```

**Ver también**:

```
 - analogReference()
- analogReadResolution()
- Tutorial: Analog Input Pins
```

## La función analogWrite()

**Descripción**:

Esta función escribe en el pin indicado un valor analógico (mediante un pulso o tren de ondas cuya duración determina el valor analógico transmitido, también llamado PWM –"Pulse Width Modulation"). Este valor puede ser utilizado para iluminar un LED con una intensidad variable o hacer rotar el eje de un motor eléctrico a velocidades variables. Cuando usamos esta función el pin generará una onda cuadrada constante de la duración especificada hasta la siguiente utilización de la función sobre el mismo pin (o una utilización de *digitalRead()* o *digitalWrite()* sobre el mismo pin). La frecuencia de la la señal PWM en la mayor parte de los pins es aproximadamente 490 Hz. En el Uno y otras placas similares, los pins 5 y 6 tienen una frecuencia de aproximadamente 980 Hz. Los pins 3 y 4 del Leonardo también funcionan a 980 Hz.

En la mayor parte de las placas Arduino (las que van equipadas con el procesador ATmega 168 o ATmega 328), esta función puede ser usada con los pins 3, 5, 6, 9, 10 y 11. En el Arduino Mega, funciona con los pins del 2 al 13 y del 44 al 46. Modelos más antiguos de Arduino con el procesador ATmega8 sólo pueden ejecutar *analogWrite()* en los pins 9, 10 y 11.

El Arduino Due puede ejecutar *analogWrite()* sobre los pins del 2 al 13 más los pins DAC0 y DAC1. Contrariamente a los pins PWM, los pins DAC0 y DAC1 son convertidores Digital -> Analógico  y actúan como verdaderas salidas analógicas (es decir, producen una salida equivalente en voltaje sin codificación PWM)

No es necesario ejecutar la función de inicialización *pinMode()* para configurar el pin como salida antes de ejecutar *analogWrite()*.

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

**Sintaxis**:

```
analogWrite(pin, valor)
```

**Parámetros**:

*pin*: el número de pin en el que queremos escribir.

*valor*: la duración del pulso: entre 0 (OFF continuo) y 255 (ON continuo)

**Devuelve**:

Nada.

**Notas**:

Las salidas PWM generadas en los pins 5 y 6 tienen una duración mayor que la esperada. Esto es debido a interacciones con las funciones *millis()*y *delay()* que usan el mismo temporizador interno usado para generar las salidas PWN. Este efecto puede ser apreciado principalmente con duraciones cortas (típicamente de 0 a 10) y puede resultar en valores de 0 que no pongan totalmente a 0 los pines 5 y 6.

**Ejemplo**:

Enciende el LED con una intensidad proporcional al valor leído en la entrada conectada con el potenciómetro.

```
int ledPin = 9;      // LED conectado al pin digital 9
```

```
int analogPin = 3;   // potenciómetro conectado al pin analógico 3
int val = 0;         // variable que almacena el valor leído (potenciómetro)

void setup()
  {
      pinMode(ledPin, OUTPUT);   // configura el the pin 9 como salida
  }
void loop()
  {
     val = analogRead(analogPin);   // lee entrada potenciómetro
     analogWrite(ledPin, val / 4);  // analogRead oscila de 0 a 1023, analogWrite entre 0 y 255
  }
```

**Ver también**:

```
- analogRead()
- analogWriteResolution()
- Tutorial: PWM
```

## FUNCIONES DE ENTRADA SALIDA DEL ARDUINO DUE

### La función analogReadResolution()

**Descripción**:

La function *analogReadResolution()* es una extensión del API analógico (Application Programme Interface) para elArduino Due. Esta función se usa para determinar el tamaño (en número de bits) del valor devuelto por la función *analogRead()*. En otras palabras: la resolución de la lectura del valor analógico efectuada en el pin. Para asegurar la compatibilidad con los modelos más antiguos (placas basadas en el procesador AVR) su valor por defecto es de 10 bits (es decir, devuelve valores entre 0 y 1023).

El Due tiene la capacidad de incrementar su capacidad ADC (Analog to Digital Converter) hasta 12 bits. Es decir, *analogRead()* puede devolver valores entre 0 y 4095.

**Sintaxis**:

```
analogReadResolution(bits)
```

**Parámetros**:

*bits*: determina la resolución (en bits) del valor devuelto por la función *analogRead()*. Este valor puede oscilar entre 1 y 32. Puedes usar resoluciones por encima de 12, pero los valores devueltos por *analogRead()*serán aproximados. Ver la nota más abajo.

**Devuelve**:

Nada

**Nota**:

Si usas *analogReadResolution()*con una resolución superior a la capacidad ADC real de tu placa, el Arduino te devolverá el resultado de la lectura con su máxima resolución posible y rellenara con ceros el resto de los bits.

Por ejemplo si usamos el Due con *analogReadResolution(16)* obtendremos una lectura de un valor aproximado de 16 bits con los primeros 12 bits conteniendo la lectura ADC realmente efectuada y los últimos 4 bits puestos a cero.

Si, por el contrario, usas *analogReadResolution()* con un valor inferior de la capacidad de tu placa, perderás resolución ya que la función te devolverá menos bits de los que el ADC de la placa obtuvo realmente.

La utilidad de usar resoluciones superiores a las que permite tu placa es que tu programa podrá ser portado a futuras placas con una resolución ADC mayor que la actual (12) sin cambiar una sóla línea de código.

**Ejemplo**:

```
void setup()
   {  Serial.begin(9600); // abrir puerta serie
   }

void loop()
   {   // leer la entrada en el pin A0 con la resolución estándar (10 bits)
       // y escribir el dato obtenido por la puerta serie
       analogReadResolution(10);
       Serial.print("ADC 10-bit (estándar) : ");
       Serial.print(analogRead(A0));

      // cambiar la resolución a 12 bits y leer entrada analógica A0
      analogReadResolution(12);
      Serial.print(", 12-bit : ");
      Serial.print(analogRead(A0));

      // cambiar la resolución a 16 bits y leer entrada analógica A0
      analogReadResolution(16);
      Serial.print(", 16-bit : ");
      Serial.print(analogRead(A0));

      // cambiar la resolución a 8 bits y leer entrada analógica A0
      analogReadResolution(8);
      Serial.print(", 8-bit : ");
      Serial.println(analogRead(A0));

      // introducir un a pequeño retraso para no inundar el monitor serie
      delay(100);
   }
```

**Ver también**:

```
- Descripción de los pins de entrada analógicos
- analogRead()
```

## La función analogWriteResolution()

**Descripcion**:

*analogWriteResolution()* es una extensión del API analógico para el Arduino Due. *analogWriteResolution()* determina la resolución de la función *analogWrite()*. Su valor por defecto es 8 bits (valores entre 0 y 255) para asegurar la compatibilidad con las placas basadas en el AVR.

El Due está equipado con:

- 12 pins funcionando en 8-bit PWM, como las antiguas placas basadas en el AVR. La resolución de estos pins se puede aumentar hasta 12 bits.
- -2 pins equipados con un DAC (Digital-to-Analog Converter) funcionado a 12 bits.

Programando la resolución de escritura a 12 bits, podemos usar *analogWrite()* con valores entre 0 y 4095 y aprovechar al máximo la resolución del DAC o usar PWM sin pérdida de resolución.

**Sintáxis**:

```
analogWriteResolution(bits)
```

**Parámetros**:

*bits*: determina la resolución (en bits) del valor devuelto por la función *analogWrite().* Este valor puede oscilar entre 1 y 32. Si se usan resoluciones superiores o inferiores a la capacidad de la placa, el valor usado en *analogWrite()* será truncado si es demasiado alto o rellenado con ceros si es demasiado baja. Ver la nota más abajo.

**Devuelve**:

Nada.

**Nota**:

Si usas *analogWriteResolution()* con una resolución superior a la capacidad ADC real de tu placa, el Arduino descartará los bits sobrantes. Por ejemplo si usamos el Due con *analogWriteResolution(16)* sobre un pin equipado con un DAC de 12 bits, solamente los primeros 12 bits serán transferidos a la función *analogWrite()* y los últimos 4 bits serán desechados.

Del mismo modo, si usas *analogWriteResolution()* con una resolución inferior a la capacidad de la placa, los bits que faltan serán rellenados con ceros. Por ejemplo, si usamos el Due con *analogWriteResolution(8)* sobre un pin dotado de un DAC de 12 bits, el Arduino añadirá 4 bits a cero a los 8 bits usados por la función *analogWrite()* para obtener los 12 bits requeridos.

**Ejemplo**:

```
void setup()
  {   // abrir puerto serie
      Serial.begin(9600);
      // configurar los pins 11, 12 y 13 como salidas
      pinMode(11, OUTPUT);
      pinMode(12, OUTPUT);
      pinMode(13, OUTPUT);
  }
void loop()
{     //Leer la entrada de A0 y escribirla en un pin PWM
      // conectado a un LED
      int sensorVal = analogRead(A0);
      Serial.print("Lectura Analógica) : ");
      Serial.print(sensorVal);

      // la resolución PWM por defecto
      analogWriteResolution(8);
      analogWrite(11, map(sensorVal, 0, 1023, 0 ,255));
      Serial.print(" , valor 8-bit PWM : ");
      Serial.print(map(sensorVal, 0, 1023, 0 ,255));

      // cambiar la resolución PWM a 12 bits
      // esta resolución se ofrece únicamente en el Due
      analogWriteResolution(12);
      analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
      Serial.print(" , valor 12-bit PWM : ");
      Serial.print(map(sensorVal, 0, 1023, 0, 4095));

      // cambiar la resolución PWM a 4 bits
      analogWriteResolution(4);
      analogWrite(13, map(sensorVal, 0, 1023, 0, 127));
      Serial.print(", valor 4-bit PWM : ");
      Serial.println(map(sensorVal, 0, 1023, 0, 127));
      delay(5);
   }
```

**Ver también**:

```
- Descripción de los pins de entrada analógica
- analogWrite()
- analogRead()
- map()
```

## FUNCIONES AVANZADAS DE ENTRADA/SALIDA

## La función tone()

**Descripcion**:

*tone()* genera una onda cuadrada de la frecuencia especificada (y un ciclo de trabajo del 50%) en un pin especificado. La duración puede ser especificada también (en caso contrario, la onda continuará hasta recibir una llamada *noTone()*. El pin puede estar conectado a un timbre piezoeléctrico o un altavoz.

Sólo podemos generar un tono a la vez. Si un tono está siendo enviado al mismo tiempo a otro pin la llamada a la función *tone()* no tendrá ningún efecto (sólo se permite un tono sobre un pin en cada momento). Si el tono está ya siendo enviado a ese pin, una llamada ulterior a *tone()* puede variar la frecuencia.

El uso de la función *tone()* puede causar interferencias con las salidas PWM en los pins 3 y 11 (salvo en la placa Mega).

No es possible generar tonos por debajo de 31 Hz. Para más detalles consultar las notas de Brett Hagman.

**Nota**:

Si quieres emitir diferentes tonos en diferentes pins (no simultáneamente sino uno tras otro) tienes que ejecutar la función *noTone()* sobre un pin antes de ejecutarla *tone()* sobre el siguiente pin.

**Sintáxis**:

```
tone(pin, frecuencia)
tone(pin, frecuencia, duración)
```

**Parámetros**:

*pin*: el pin en el cual se quiere general el tono

*frecuencia*: la frecuencia del tono en herzios – tipo: *unsigned int*

duración: la duración del tono en milisegundos (opcional) – tipo: *unsigned long*

**Devuelve**:

Nada

**Ver también**:

```
- noTone()
- analogWrite()
- Tutorial:Tone
- Tutorial:Pitch follower
- Tutorial:Simple Keyboard
- Tutorial: multiple tones
- Tutorial: PWM
```

## La función noTone()

**Descripción**

Termina la generación de la onda cuadrada que fue comenzada por una función *tone()*. No tiene ningún efecto si no se ha ejecutado ninguna función *tone()* sobre ese pin antes.

Nota: Si quieres emitir diferentes tonos en diferentes pins (no simultáneamente sino uno tras otro) tienes que ejecutar la función *noTone()* sobre un pin antes de ejecutarla *tone()* sobre el siguiente pin.

**Sintáxis**:

```
noTone(pin)
```

**Parámetros**:

*pin*: el pin sobre el cual se quiere detener la generación de tono.

**Devuelve**:

Nada

**Ver también**:

```
- tone()
```

## La función shiftOut()

**Descripción**

Emite un byte bit a bit por el pin indicado. Se puede emitir empezando por el bit de mayor peso (el primero por la izquierda) o el de menor peso (el primero por la derecha). Cada bit se emite por riguroso orden en el pin indicado. Una vez que el bit está disponible en el pin, el pin de reloj (clock pin) conmuta (alta al comienzo, baja a continuación) para indicar que el bit está disponible en pin de salida.

Nota: si estamos comunicando con un dispositivo que funciona con subidas de nivel en el reloj (de baja a alta), necesitarás asegurarte que el pin de reloj está en baja antes de ejecutar la función *shiftOut()* con una llamada a la función *digitalWrite(clockPin, LOW)*.

Nota: La función *shiftOut()* es una implementación de software; si quieres una implementación más rápida (pero que sólo funciona en ciertos pins) la librería SPI proporciona una implementación hardware de esta funcionalidad.

**Sintaxis**:

```
shiftOut(dataPin, clockPin, bitOrder, valor)
```

**Parámetros**:

*dataPin*: el pin que será usado para emitir cada bit. Tipo: *int*

*clockPin*: el pin usado para enviar la señal de conmutación una vez que el *dataPin* está emitiendo el bit correcto. Tipo: *int*

*bitOrder*: el orden en que los bits serán emitidos; sea MSBFIRST (empezando por el de mayor peso y siguiendo hacia la derecha) o LSBFIRST (comenzando por el de menor peso y siguiendo hacia la izquierda).

valor: el octeto que será emitido. Tipo: *byte*

**Devuelve**:

Nada

**Notas**:

Los pins *dataPin* y *clockPin* deben de ser configurados como salidas mediante una llamada a la función *pinMode()* antes de ejecutar *shiftOut()*.

*shiftOut* puede emitir sólamente un byte (octeto = 8 bits) cada vez que es ejecutada, por lo tanto se necesitan dos iteraciones para emitir valores mayores que 255.

```
// Para comenzar con el bit de más peso
int data = 500;
// emitir el byte de más peso
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// y ahora el de menos peso
shiftOut(dataPin, clock, MSBFIRST, data);
```

*Y este sketch para comenzar con el bit de menor peso*

```
data = 500;
// emitir el byte de menos peso
```

```
shiftOut(dataPin, clock, LSBFIRST, data);
// y ahora el de más peso
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

**Ejemplo**:

Para el circuito que acompaña a este ejemplo ver el tutorial "controlling a 74HC595 shift register".

```
 //*************************************************************//
//  Name    : shiftOutCode, Hello World                        //
//  Author  : Carlyn Maw,Tom Igoe                              //
//  Date    : 25 Oct, 2006                                     //
//  Version : 1.0                                              //
//  Notes   : Code for using a 74HC595 Shift Register          //
//          : to count from 0 to 255                           //
//*************************************************************
//Pin connected to ST_CP of 74HC595
int latchPin = 8;
//Pin connected to SH_CP of 74HC595
int clockPin = 12;
////Pin connected to DS of 74HC595
int dataPin = 11;

void setup()
  {   //set pins to output because they are addressed in the main loop
      pinMode(latchPin, OUTPUT);
      pinMode(clockPin, OUTPUT);
      pinMode(dataPin, OUTPUT);
  }

void loop()
   {   //count up routine
       for (int j = 0; j < 256; j++)
        { //ground latchPin and hold low for as long as you are transmitting
           digitalWrite(latchPin, LOW);
          shiftOut(dataPin, clockPin, LSBFIRST, j);
          //return the latch pin high to signal chip that it
          //no longer needs to listen for information
          digitalWrite(latchPin, HIGH);
          delay(1000);
        }
   }
```

**Ver también**:

```
- shiftIn()
- SPI
```

## La función shiftIn()

**Descripción**:

Esta es la función inversa de la anterior (*shiftOut*). La función *shiftIn()* lee un byte bit a bit del pin de entrada especificado. Como en la función anterior (*shiftout*) podemos especificar el sentido de lectura (de izquierda a derecha o viceversa). Cada operación de lectura se realiza así: cuando el pin reloj ("clock pin") se pone en alta (HIGH) se procede a leer el bit en el pin, a continuación el pin reloj se pone en baja tensión (LOW).

Si estamos comunicando con un dispositivo que funciona con subidas de nivel en el reloj (de baja a alta), necesitarás asegurarte que el pin de reloj está en baja antes de ejecutar la función *shiftIn()* con una llamada a la función *digitalWrite(clockPin, LOW)*.

La función *shiftIn()* es una implementación de software; si quieres una implementación más rápida (pero que sólo funciona en ciertos pins) la librería SPI proporciona una implementación hardware de esta funcionalidad.

**Sintaxis**:

```
byte leido = shiftIn(dataPin, clockPin, bitOrder)
```

**Parámetros**:

*dataPin*: el pin que será usado para leer cada bit. Tipo: *int*

*clockPin*: el pin usado para enviar la señal de conmutación una vez que el *dataPin* ha leido el bit correcto. Tipo: *int*

*bitOrder*: el orden en que los bits serán emitidos; sea MSBFIRST (empezando por el de mayor peso y siguiendo hacia la derecha) o LSBFIRST (comenzando por el de menor peso y siguiendo hacia la izquierda). (Most Significant Bit First, or, Least Significant Bit First

Devuelve:

Nada

**Ver también**:

```
- shiftOut()
- SPI
```

## La función pulseIn()

**Descripción**:

Lee un pulso (HIGH o LOW) de un pin determinado. Por ejemplo, si el valor especificado como parámetro es HIGH, *pulseIn()* espera a que el pin esté en HIGH, en ese momento comienza la cuenta (timing) hasta que la tensión en el pin esté en LOW momento en que cesa la cuenta. *pulseIn()* devuelve la duración del pulso en microsegundos. Asimismo, *pulseIn()* devuelve control con un cero como return si el pulso no comienza dentro de un periodo especificado (time out)

El resultado de esta función ha sido determinado de manera empírica y probablemente muestra errores cuando los pulsos son demasiado largos. pulseIn() funciona con pulsos desde 10 microsegundos hasta 3 minutos de duración.

**Sintaxis**:

```
pulseIn(pin, value)
pulseIn(pin, value, timeout)
```

**Parámetros**:

*pin*: El número del pin del que se quiere leer el la duración del pulso. Tipo: *int*

*value:* El tipo de pulso que se quiere leer (HIGH o LOW). Tipo: *int*

*timeout* (opcional): el número de microsegundos que se quiere esperar para que comienze el pulso; por defecto es un segundo. Tipo: *unsigned long*

**Devuelve**:

La duración del pulso (en microsegundos) o 0 si el pulso no comenzó antes del tiempo límite (*time out*). Tipo: *unsigned long*

Ejemplo:

```
int pin = 7;
unsigned long duration;
void setup()
   {  pinMode(pin, INPUT);
   }
void loop()
   {
      duration = pulseIn(pin, HIGH);
   }
```

# FUNCIONES DE TIEMPO

## La función millis()

**Descripción**:

Esta función devuelve el número de milisegundos transcurridos desde que el Arduino comenzó a ejecutar el programa en curso. Este número será puesto a cero de nuevo cada 50 días (aproximadamente).

**Parámetros**:

Ninguno

**Devuelve**:

Número de milisegundos transcurridos desde que el Arduino comenzó a ejecutar el programa en curso. Tipo: *unsigned long*

**Ejemplo**:

```
unsigned long time;
void setup()
   {  Serial.begin(9600);
   }
void loop()
   {  Serial.print("Tiempo: ");
      time = millis();
      //muestra el tiempo transcurrido desde que comenzó el programa
      Serial.println(time);
      // espera un Segundo para no inundar la pantalla de datos.
      delay(1000);
   }
```

**Nota**:

Ten en cuenta que *millis()* devuelve un *unsigned long*, se producirán errores si tratas a este número como si tuviera otro tipo (por ejemplo *int*).

**Ver también**:

```
- micros()
- delay()
- delayMicroseconds()
- Tutorial: Blink Without Delay
```

## La función micros()

**Descripción**:

Esta función devuelve el número de microsegundos transcurridos desde que el Arduino comenzó a ejecutar el programa en curso. Este número será puesto a cero de nuevo cada 70 minutos (aproximadamente). En las placas que funcionan a 16 MHz (por ejemplo el Duemilanove y el Nano), esta función tiene una resolución de 4 microsegundos (es decir, el valor devuleto es siempre un múltiplo de 4). En placas Arduino funcionando a 8 MHz (por ejemplo el LilyPad), esta función tiene una resolución de 8 microsegundos.

**Parámetros**:

Ninguno

**Devuelve**:

Número de microsegundos transcurridos desde que el Arduino comenzó a ejecutar el programa en curso. Este número será puesto a cero de nuevo cada 70 minutos (aproximadamente). Tipo: *unsigned long.*

Nota: un milisegundo tiene 1,000 microsegundos y un segundo 1,000,000 de microsegundos.

**Ejemplo**:

```
unsigned long time;
void setup()
   {   Serial.begin(9600);
   }
void loop()
   {   Serial.print("Tiempo: ");
       time = micros();  // muestra el tiempo transcurrido desde que comenzó el programa
       Serial.println(time);
       // espera un Segundo para no inundar la pantalla de datos.
       delay(1000);
   }
```

**Ver también**:

```
- millis()
- delay()
- delayMicroseconds
```

## La función delay()

**Description**:

Introduce una pausa en el programa de una duración especificada (en milisegundos) como parámetro. (1 segundo = 1000 milisegundos)

**Sintaxis**:

```
delay(ms)
```

**Parámetros**:

*ms*: la duración de la pausa en  número de milisegundos. Tipo: *unsigned long*

**Devuelve**:

Nada.

**Ejemplo**:

```
int ledPin = 13;                // LED conectado al pin digital13
void setup()
   {   pinMode(ledPin, OUTPUT);      // configura el pin digital como salida
   }

void loop()
   {
      digitalWrite(ledPin, HIGH);   // enciende el LED
      delay(1000);                  // pausa de 1 segundo
      digitalWrite(ledPin, LOW);    // apaga el LED
      delay(1000);                  // otra pausa de 1 segundo
  }
```

**Nota**:

Es relativamente fácil programar un LED intermitente con la función *delay().* De hecho muchos sketches usan esta función con similares propósitos.

Sin embargo el uso de *delay()* tiene algunos inconvenientes. Por ejemplo, durante el tiempo que el sketch está es estado de pausa no se puede

leer la entrada de sensores, ni realizar cálculos o manipular pins. En otras palabras la función *delay()* prácticamente paraliza toda actividad en el Arduino.

Para paliar este problema los programadores usan la función *millis()* (ver el sketch citado más abajo). Sin embargo, las interrupciones no quedan bloqueadas por la función *delay()*. Las comunicaciones en serie vía el pin RX quedan registradas y los valores y estados de los pins PWM (*analogWrite*) se mantiene durante la pausa.

**Ver también**:

```
- millis()
- micros()
- delayMicroseconds()
- Blink Without Delay example
```

## La función delayMicroseconds()

**Descripción**:

Esta función detiene la ejecución del programa durante el tiempo (en microsegundos) indicado por el parámetro.

De momento el valor máximo para el parámetro tiempo es de 16383 microsegundos (aunque esto puede cambiar en futuras versiones del Arduino). Para pausas de más de unos cuantos miles de microsegundos se debe usar *delay()* en vez de esta función.

**Sintaxis**:

```
delayMicroseconds(us)
```

**Parámetros**:

*us*: el número de microsegundos que dura la pausa. Tipo: *unsigned int*

**Devuelve**:

Nada.

**Ejemplo**:

```
int outPin = 8;                 // pin digital 8
void setup()
   {
      pinMode(outPin, OUTPUT);      // configura el pin digital como salida
   }

void loop()
   {
      digitalWrite(outPin, HIGH);   // Pone el pin en alta
      delayMicroseconds(50);        // Pausa de 50 microsegundos
      digitalWrite(outPin, LOW);    // Pone el pin en baja
      delayMicroseconds(50);        // Pausa de 50 microsegundos
   }
```

Este sketch configura el pin número 8 como un pin de salida y envía un tren de pulsos con un periodo de 100 microsegundos.

**Problemas conocidos**:

Esta function es muy fiable a partir de 3 microsegundos. No se garantiza que *delayMicroseconds* funcione de forma precisa con tiempos inferiores a éste.

A partir del modelo0018 del Arduino, *delayMicroseconds()* no desactiva las interrupciones.

**Ver también**:

```
- millis()
```

```
- micros()
- delay()
```

## FUNCIONES MATEMÁTICAS

### La función min(x, y)

**Descripción**:

Calcula el mínimo de dos números.

**Parámetros**:

*x*: el primer número (de cualquier tipo)

*y*: el segundo número (de cualquier tipo)

**Devuelve**:

El menor de los dos números.

**Ejemplo**:

```
sensVal = min(sensVal, 100); // asigna a la variable sensVal el menor de sensVal y 100
            // sirve también para asegurarse que la variable nunca supera el valor 100.
```

**Nota**:

Por extraño que parezca, la función *max()* se usa a menudo para limitar el valor mínimo de una variable y la *min()* para limitar el máximo.

**Advertencia**:

Debido a la manera en que la función *min()* ha sido implementada es recomendable evitar usar otras funciones dentro de los paréntesis conteniendo los parámetros. Por ejemplo:

*min(a++, 100);* // es a evitar ya que devuelve resultados incorrectos.

Esta otra manera de usarla es, en cambio, más segura:

a++;

min(a, 100); // usar esta modalidad. ¡No incluir funciones dentro del paréntesis!

**Ver también**:

```
- max()
- constrain()
usando
```

### La función max(x, y)

Descripción:Calcula el máximo de dos números.

**Parámetros**:

*x*: el primer número (de cualquier tipo)

*y*: el segundo número (de cualquier tipo)

**Devuelve**:

El mayor de los dos números.

**Ejemplos**:

```
sensVal = max(sensVal, 20); // asigna a la variable sensVal el mayor de sensVal y 20
            // sirve también para asegurarse que la variable nunca es inferior a 20
```

**Nota**:

Por extraño que parezca, la función *max()* se usa a menudo para limitar el valor mínimo de una variable y la *min()* para limitar el máximo.

**Advertencia**:

Debido a la manera en que la función *max()* ha sido implementada es recomendable evitar usar otras funciones dentro de los paréntesis conteniendo los parámetros. Por ejemplo:

*max(a–, 100);* // es a evitar ya que devuelve resultados incorrectos.

Esta otra manera de usarla es, en cambio, más segura:

a–;

min(a, 100); // usar esta modalidad. ¡No incluir funciones dentro del paréntesis!

**Ver también**:

```
- min()
- constrain()
```

## La función abs(x)

**Descripción**:

Calcula el valor absoluto de un número.

**Parámetros**:

*x*: el número

**Devuelve**:

*x*: si *x* es mayor o igual que 0.

*-x*: si *x* es menor que 0.

**Advertencia**:

Debido a la manera en que la función *abs()* ha sido implementada es recomendable evitar usar otras funciones dentro de los paréntesis conteniendo los parámetros. Por ejemplo:

abs(a++); // es a evitar ya que devuelve resultados incorrectos.

a++; // usa esta modalidad

abs(a); //¡no incluir funciones dentro de abs()!

## La función constrain(x, a, b)

**Descripción**:

Limita a un número a permanecer dentro de un rango.

**Parámetros**:

*x*: El número a limitar (de cualquier tipo)

*a*: El límite inferior (de cualquier tipo)

*b*: El límite superior (de cualquier tipo).

**Devuelve**:

*x*: si *x* está entre *a* y *b*

*a*: si *x* < *a*

b: si x > b

**Ejemplo**:

```
sensVal = constrain(sensVal, 10, 150);
// limita el range de valores leidos del sensor entre 10 y 150
```

**Ver también**:

```
- min()
- max()
```

## La función map(value, fromLow, fromHigh, toLow, toHigh)

**Descripción**:

Trasforma un número de una escala a otra. Es decir, un valor ("*value*") igual a "*fromLow*" se transformaría en "*toLow*" y un valor igual a "*fromHigh*" se transformaría en "*toHigh*". Los valores dentro del rango *fromLow-fromHigh* se transformarían proporcionalmente en valores dentro del rango *toLow-toHigh*.

Esta función no limita a los valores a estar dentro del rango (si es esto lo que quieres conseguir, debes de usar la función *constrain()*). En efecto, a veces es útil trabajar con valores fuera de rango.

El límite inferior de cualquiera de los dos rangos puede ser menor o mayor que el límite superior. De hecho esto puede ser muy útil cuando queremos usar la función *map()* para invertir un rango. Por ejemplo para transformar un valor de un rango creciente a otro decreciente, como por ejemplo:

*y = map(x, 1, 50, 50, 1);*

La función *map()* puede manejar también rangos que empiezan o terminan en la zona negativa, como por ejemplo:

*y = map(x, 1, 50, 50, -100);*

La función *map()* trabaja con números enteros y por lo tanto no genera números fraccionarios. Los restos fraccionarios son truncados (no redondeados).

**Parámetros**:

*value*: el valor a transformar de escala.

*fromLow*: el límite inferior del rango origen

*fromHigh*: el límite superior del rango origen

*toLow*: el límite inferior del rango destino

*toHigh*: el límite superior del rango destino

**Devuelve**:

El valor transformado en el rango de destino.

**Ejemplo**:

```
/* Transforma un valor analógico (entre 0 y 1023) a un valor en 8 bits (0 to 255) */
void setup() {}
void loop()
   {      int val = analogRead(0);
          val = map(val, 0, 1023, 0, 255);
          analogWrite(9, val);
   }
```

**Apéndice:**

Para los aficionados a las matemáticas, ahí va el código de esta función:

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
   {  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
   }
```

**Ver también**:

– *constrain()*

## La función pow(base, exponent)

**Descripción**

Calcula el valor de un número elevado a una potencia. *Pow()* puede ser usado también para elevar un número a una potencia fraccional. Esto es muy útil para generar los valores de una curva (que generalmente se expresa en términos de exponentes fraccionales).

**Parámetros**:

*base*: el número. Tipo: *float*

*exponent*: la potencia a la que la base es elevada. Tipo: *float*

**Devuelve**:

El resultado de la exponenciación. Tipo: *double*

**Ejemplo**:

Ver la función “fscale” de la librería “*code*”.

**Ver también**:

– *sqrt()*

– *float*

– *double*

## La función sqrt(x)

**Descripción**:

Calcula la raíz cuadrada de un número.

**Parámetros**:

*x*: el número (de cualquier tipo)

**Devuelve**:

La raíz cuadrada del número. Tipo: *double*

Ver también:

> *- pow()*
> *- sq()*

### La función sin(rad)

**Descripción**:

Calcula el seno de un ángulo (expresado en radianes). El resultado estará comprendido entre -1 y 1.

**Parámetros**:

*rad*: el ángulo en radianes. Tipo: *float*

**Devuelve**:

El seno del ángulo. Tipo: *double*

**Ver también**:

> *- cos()*
> *- tan()*
> *- float*
> *- double*

### La función cos(rad)

**Descripción**:

Calcula el coseno de un ángulo (expresado en radianes). El resultado estará comprendido entre -1 y 1.

**Parámetros**:

*rad*: el ángulo en radianes. Tipo: *float*

**Devuelve**:

El coseno del ángulo. Tipo: *double*

Ver también:

> *-sin()*
> *-tan()*
> *-float*
> *-double*

### La función tan(rad)

**Descripción**:

Calcula la tangente de un ángulo (expresado en radianes). El resultado estará comprendido entre -∞ y +∞.

**Parámetros**:

*rad*: el ángulo en radianes. Tipo: *float*

**Devuelve**:

La tangente del ángulo. Tipo: *double*

**Ver también**:

> *-sin()*
> *-cos()*
> *-float*
> *-double*

## FUNCIONES ALEATORIAS

### La función randomSeed(seed)

**Descripción**:

*randomSeed()* sirve para inicializar el pseudo generador de números aleatorios. El parámetro le dice a la función en que punto de la secuencia debe de comenzar. Esta secuencia, que es muy larga y aleatoria es siempre la misma.

En caso de que necesitemos que la secuencia de números generados por la función *random()* sea diferente en diferentes ejecuciones del mismo sketch se recomienda el uso de *randomSeed()* para inicializar el generador de números aleatorios con una entrada (*seed*) más o menos aleatoria (como por ejemplo la lectura de *analogRead()* en un pin libre).

Por otra parte, puede ser interesante en ciertas ocasiones disponer de series de números pseudo aleatorios que se repiten exactamente de la misma manera. Esto puede conseguirse llamando a *randomSeed()* con número fijo antes de comenzar la secuencia aleatoria.

**Parámetros**:

*seed*: número de base para establecer la secuencia. Tipo: *long, int*

**Devuelve**:

Nada

**Ejemplo**:

```
long randNumber;
void setup()
   {  Serial.begin(9600);
      randomSeed(analogRead(0));
   }

void loop()
   {  randNumber = random(300);
      Serial.println(randNumber);
      delay(50);
   }
```

**Ver también**:

> *- random*

## La función random()

**Descripción**:

La función *random()* general números pseudo-aleatorios.

**Sintaxis**:

```
random(max)
random(min, max
```

**Parámetros**:

*min* – límite inferior del valor aleatorio, inclusive (opcional)

*max* – límite superior de valor aleatorio, exclusive

**Devuelve**:

Un número aleatorio entre *min* y *max*-1 (tipo: *long*)

**Nota**:

En caso de que necesitemos que la secuencia de números generados por la función *random()* sea diferente en diferentes ejecuciones del mismo sketch se recomienda el uso de *randomSeed()* para inicializar el generador de números aleatorios con una entrada (*seed*) más o menos aleatoria (como por ejemplo la lectura de *analogRead()* en un pin libre).

Por otra parte, puede ser interesante en ciertas ocasiones disponer de series de números pseudo aleatorios que se repiten exactamente de la misma manera. Esto puede conseguirse llamando a *randomSeed()* con número fijo antes de comenzar la secuencia aleatoria.

**Ejemplo**:

```
long randNumber;

void setup()
   { Serial.begin(9600);
     // si el pin de entrada analógica 0 no está conectado, el ruido aleatorio analógico
     // origina que la llamada a randomSeed() genere
     // números de inicialización (seed) diferentes cada vez que se ejecute el sketch.
     // randomSeed() hará que la función random() devuelva diferentes secuencias.

     randomSeed(analogRead(0));
   }

void loop()
   { // imprimir un número aleatorio comprendido entre 0 y 299
      randNumber = random(300);
      Serial.println(randNumber);
      // imprimir un número aleatorio comprendido entre 10 y 19
      randNumber = random(10, 20);
      Serial.println(randNumber);
      delay(50);
   }
```

**Ver también**:

```
- randomSeed()
```

## FUNCIONES CON BITS Y BYTES

## La función lowByte()

**Descripción**:

Extrae el octeto (byte) de menor peso de una variable (el byte más a la derecha).

**Sintaxis**:

```
lowByte(x)
```

**Parámetros**:

*x*: un valor de cualquier tipo.

**Devuelve**:

Un byte

**Ver también**:

– *highByte()*

– *word()*

## La función highByte()

**Descripción**:

Extrae el octeto (byte) de mayor peso de una palabra (o el segundo byte de menor peso de un tipo de datos de mayor tamaño).

**Sintaxis**:

```
highByte(x)
```

**Parámetros**:

*x*: un valor de cualquier tipo.

**Devuelve**:

Un byte

**Ver también**:

```
- lowByte()
- word()
```

## La función bitRead()

**Descripción**:

Lee un bit de un número en una posición especificada.

**Sintáxis**:

```
bitRead(x, n)
```

**Parámetros**:

*x*: el número delque se lee el bit.

*n*: la posición de bit que se quiere leer dentro del número. Comenzando por la derecha (posición 0)

**Devuelve**:

El valor del bot (0 or 1).

**Ver también**:

```
- bit()
- bitWrite()
- bitSet()
- bitClear()
```

## La función bitWrite()

**Descripción**:

Escribe un bit en una variable numérica en una posición especificada.

**Sintaxis**

```
bitWrite(x, n, b)
```

**Parámetros**:

*x*: La variable numérica en la cual se desea escribir.

*n*: La posición de bit en la que se quiere escribir dentro del número. Comenzando por la derecha (posición 0).

*b*: El valor que se quiere escribir en esa posición de bit (0 o 1)

**Devuelve**:

Nada

**Ver también**:

```
- bit()
- bitRead()
- bitSet()
- bitClear()
```

## La función bitSet()

Descripción:

Pone a 1 el bit especificado en una variable.

**Sintaxis**:

```
bitSet(x, n)
```

**Parámetros**:

*x*: La variable numérica en la cual se desea poner a 1 un bit especificado.

*n*: La posición del bit que se quiere poner a 1. Comenzando por la derecha (posición 0).

**Devuelve**:

nada

**Ver también**:

```
- bit()
- bitRead()
- bitWrite()
- bitClear()
```

## La función bitClear()

Descripción:

Pone a 0 el bit especificado en una variable.

Sintaxis:

```
bitClear(x, n)
```

**Parámetros**:

*x*: La variable numérica en la cual se desea poner a 0 un bit especificado.

*n*: La posición del bit que se quiere poner a 0. Comenzando por la derecha (posición 0).

**Devuelve**:

nada

**Ver también:**

```
- bit()
- bitRead()
- bitWrite()
- bitSet()
```

## La función bit()

**Descripción**:

Calcula el valor del bit especificado (el bit 0 vale 1, el bit 1 vale 2, el bit 2 vale 4, etc.).

**Sintaxis**:

```
bit(n)
```

**Parámetros**:

*n*: la posición del bit cuyo valor queremos calcular

**Devuelve**:

El valor del bit especificado

**Ver también**:

```
- bitRead()
- bitWrite()
- bitSet()
- bitClear()
```

## INTERRUPCIONES EXTERNAS

### La función attachInterrupt()

Descripción:

Esta función determina la rutina (programa) que se ha de ejecutar cuando se presenta una interrupción. La ejecución de *attachInterrupt()* reemplaza cualquier otra función de interrupción que estuviera asociada con esa interrupción. La mayor parte de las placas Arduino tiene dos entradas para interrupciones externas: la número 0 (asociada con el pin digital No 2) y la número 1 (asociada al pin digital número 3). La tabla de abajo muestra los pins disponibles para interrupciones en las diferentes placas:

| Board | int.0 | int.1 | int.2 | int.3 | int.4 | int.5 |
|---|---|---|---|---|---|---|
| UnoEthernet | 2 | 3 | | | | |
| Mega2560 | 2 | 3 | 21 | 20 | 19 | 18 |
| Leonardo | 3 | 2 | 0 | 1 | 7 | |

El Arduino Due tiene muchas más posibilidades ya que nos permite asociar una función de interrupción a cualquier pin disponible. Para hacerlo basta con especificar directamente el número de pin en la función *attachInterrupt()*.

**Nota**:

Dentro de la función asociada (ISR = Interrupt Service Routine) la función *delay()* no puede funcionar y el valor devuelto por *millis()* no se incrementará. Los datos recibidos por la puerta serie mientras la rutina ISR es ejecutada se perderán. Todas las variables que deban de ser modificadas por la rutina ISR deben de ser declaradas como "*volatile*". Ver las explicaciones sobre las rutinas ISR a continuación.

**Usando Interrupciones**:

Las interrupciones son una técnica muy útil para que ciertas cosas se hagan automáticamente (por ejemplo para detectar entradas de usuario o leer variaciones en un codificador angular y además pueden ayudar a resolver problemas de temporización y sincronización.

Si, por ejemplo, quieres activar el procesador del Arduino solamente cuando suceda un cierto evento: un sensor que detecta la caída de una moneda, o una entrada desde el teclado, o un cambio en el pulso emitido por un codificador angular, el uso de interrupciones puede liberar al microprocesador para realizar otras tareas mientras no se presenta la interrupción (en vez de estar muestreando todo el tiempo para ver el estado de un determinado pin).

**Rutinas de Interrupción (ISR = "Interrupt Service Routines")**:

Las rutinas ISRs son funciones especiales que tienen algunas limitaciones con respecto a una función normal. Por ejemplo, una ISR no puede tener parámetros y no devuelve nada.

Generalmente una ISR debe de ser tan corta y rápida como posible. Si tu sketch usa varias ISR, sólo una puede ser ejecutada a la vez, todas las demás interrupciones serán desactivadas hasta que la que está en curso haya sido tratada.

Como las funciones Como las funciones *delay()* y *millis()* están basadas en el uso de interrupciones, no pueden funcionar mientras una ISR esté siendo ejecutada. Sin embargo *delayMicroseconds()*, que no utiliza interrupciones, funcionará normalmente.

Normalmente se usan variables gobales para pasar datos entre la ISR y el programa principal. Para asegurarse de que las variables usadas dentro de una ISR son actualizadas correctamente hay que declararlas como "*volatile*".

Para más información sobre interrupciones ver las notas de Nick Gammon sobre este tema.

**Sintaxis**:

```
attachInterrupt(interrupt, ISR, mode)
attachInterrupt(pin, ISR, mode)  (sólo en el Arduino Due)
```

**Parámetros**:

*interrupt*: el número de interrupción. Tipo: *int*

*pin*: el número de pin (sólo en el Arduino Due only)

*ISR*: La rutina ISR que debe de ser llamada cuando esta interrupción se active; esta función no usa parámetros y no devuelve ningún valor.

*mode*: define cuándo se activa la interrupción. Hay cuatro casos predefinidos:

- LOW activa la interrupción siempre que el pin esté en baja (LOW)
- CHANGE activa la interrupción cuando el pin cambia de estado (de alta a baja o viceversa)
- RISING activa la interrupción cuando el pin pasa de baja a alta (low -> high)
- FALLING activa la interrupción cuando el pin pasa de alta a baja (high -> low)

El Due permite también:

- HIGH activa la interrupción siempre que el pin esté en alta (HIGH).(Arduino Due only)

**Devuelve**:

Nada

**Ejemplo**:

```
int pin = 13;
volatile int state = LOW;
void setup()
   {   pinMode(pin, OUTPUT);
       attachInterrupt(0, blink, CHANGE);
   }
void loop()
   {   digitalWrite(pin, state);
   }
void blink()
   {   state = !state;
   }
```

**Ver también:**

```
- detachInterrupt
```

## La función detachInterrupt()

**Descripción**:

Desactiva una interrupción dada.

**Sintaxis**:

```
detachInterrupt(interrupt)
```

```
detachInterrupt(pin)  (sólo en el Arduino Due)
```

**Parámetros**:

*interrupt*: el número de interrupción a desactivar (ver *attachInterrupt()* para más información).

*pin*: el número de number de la interrupción a desativar (sólo en el Arduino Due)

**Ver también**:

```
- attachInterrupt()
```

## FUNCIONES DE INTERRUPCIÓN

### La función interrupts()

**Descripción**:

Esta función vuelve a activar las interrupciones (tras que éstas hayan sido desactivadas por la función *noInterrupts()*). Las interrupciones permiten la ejecución de ciertas tareas importantes mientras el procesador ejecuta un sketch y están activadas por defecto.

Algunas funciones no podrán ser llamadas mientras las interrupciones estén desactivadas (porque utilizan interrupciones) y las comunicaciones de entrada no serán leídas. Las interrupciones pueden incrementar el tiempo de ejecución del código principal. Por esta razón puede ser recomendable desactivarlas durante la ejecución de partes particularmente importantes del código de nuestro sketch.

**Parámetros**:

Ninguno

**Devuelve**:

Nada

**Ejemplo**:

```
void setup() {}

void loop()
   {  noInterrupts();
      //incluir aquí el código crucial que no puede verse interrumpido.
      interrupts();
      //el resto del código aquí
   }
```

Ver también:

```
- noInterrupts()
- attachInterrupt()
- detachInterrupt()
```

### La funciónn olinterrupts()

**Descripción**:

Esta función desactiva las interrupciones (puedes activarlas con la función *interrupts()* que acabamos de ver). Las interrupciones permiten la ejecución de ciertas tareas importantes mientras el procesador ejecuta un sketch y están activadas por defecto.

Algunas funciones no podrán ser llamadas mientras las interrupciones estén desactivadas (porque utilizan interrupciones) y las comunicaciones de entrada no serán leídas. Las interrupciones pueden incrementar el tiempo de ejecución del código principal.

Por esta razón puede ser recomendable desactivarlas durante la ejecución de partes particularmente importantes del código de nuestro sketch.

**Parámetros**:

Ninguno

**Devuelve**:

Nada

**Ejemplo**:

```
void setup() {}
void loop()
   {  noInterrupts();
      //incluir aquí el código crucial que no puede verse interrumpido.
      interrupts();
      //el resto del código aquí
   }
```

**Ver también**:

```
- interrupts()
- attachInterrupt()
- detachInterrupt()
```

FUNCIONES DE COMUNICACIONES

## Funciones de comunicaciones en serie (serial)

Los puertos de comunicación serie se usan para la comunicación entre el Arduino y un ordenador u otros dispositivos. Todos los Arduinos tienen por lo menos un puerto serie (también conocido como UART o USART). El Arduino usa los pins digitales 0 (RX) y 1 (TX) como puerto serie (llamado "***serial***").

También se puede usar la salida USB para comunicar en serie con el ordenador vía el USB. Recuerda que si utilizas la función *serial* no puedes usar los pins 0 y 1 como entrada o salida digital. (RX es el pin de recepción y TX en de transmisión)

Puedes usar el monitor serie incorporado en el ambiente de programación del Arduino para comunicar con un Arduino. Haz clic el botón serial del monitor en la barra de herramientas y selecciona la misma velocidad usada con la función *begin()*.

El Arduino Mega tiene tres puertos serie adicionales: ***Serial1*** en los pins 19 (RX) y 18 (TX), ***Serial2*** en los pins 17 (RX) y 16 (TX), **Serial3** en los pins 15 (RX) y 14 (TX). Para utilizar estos pins para comunicar con tu ordenador, necesitarás un adaptador USB-serie adicional, pues no están conectados con el adaptador USB-serie del Mega.

Si los quieres usar para comunicar con un dispositivo serial externo TTL, conecta el pin de TX con el pin del RX del dispositivo externo, el RX con el pin del TX del dispositivo externo, y la tierra del mega con la tierra del dispositivo externo. (importante: no conectes estos pins directamente con un puerto serie RS232 ya estos funcionan a +/- 12V y pueden dañar tu Arduino).

El Arduino Due tiene tres puertos serie TTL a 3.3V adicionales: ***Serial1*** en los pins 19 (RX) y 18 (TX); ***Serial2*** en los pins 17 (RX) y 16 (TX), ***Serial3*** en los pins 15 (RX) y 14 (TX). Los pins 0 y 1 también están conectados con los pins correspondientes del microprocesador serial de ATmega16U2 USB-TTL, que está conectado con el puerto "*debug*" del USB. Además, hay un puerto serie USB nativo en el microprocesador de SAM3X.

El Arduino Leonardo utiliza ***Serial1*** para comunicar vía serie TTL #5V# en los pins 0 (RX) y 1 (TX). ***Serial*** está reservado para la comunicación de la CDC del USB. Para más información, acceda a la documentación del Leonardo.

## La función if(Serial)

**Description**:

Esta función (introducida con el Arduino 1.0.1) indica si el puerto serie especificado está activado.

En el Leonardo esta función indica si la conexión serie USB CDC está activada. En todos los demás casos, incluyendo *if(Serial1)* en el Leonardo, esta función devolverá siempre "*true*".

**Sintaxis**:

Para todos los modelos:

```
if (Serial)
```

Para el Arduino Leonardo:

```
if (Serial1)
```

Para el Arduino Mega:

```
if (Serial1)
if (Serial2)
if (Serial3)
```

**Parámetros**:

ninguno

**Devuelve**:

Un valor tipo *boolean* : "true" si el puerto serie especificado está disponible. En el Leonardo esta función sólo devolverá "false" si la conexión serie USB CDC no está activada.

**Ejemplo**:

```
void setup()
   {   //Initializar puerto serie y esperar a que esté activado:
       Serial.begin(9600);
       while (!Serial)
          {          ; //esperar a que el puerto se abra. Necesario sólo en el Leonardo
         }
   }

void loop()
   {     //aqui va el programa de comunicaciones…
   }
```

**Ver también**:

```
- begin()
- end()
- available()
- read()
- peek()
- flush()
- print()
- println()
- write()
- SerialEvent()
```

## La función available()

Descripción:

Esta función devuelve el número de bytes (caracteres u octetos) disponibles para su lectura desde el puerto serie. Estos son los bytes que han llegado ya al puerto serie y ha sido almacenados en el buffer de recepción en serie (que tiene una capacidad total de 64 bytes). La función *available()* hereda de la clase Stream.

**Sintaxis**:

```
Serial.available()
```

Para l Arduino Mega solamente:

```
Serial1.available()
Serial2.available()
Serial3.available()
```

**Parámetros**:

none

**Devuelve**:

El número de bytes disponibles para su lectura.

**Ejemplo**:

```
int incomingByte = 0;   // para datos llegando por el puerto serie.

void setup()
   {   Serial.begin(9600);   // abre el Puerto serie y la configura a 9600 bps
   }

void loop()
   {    // enviar datos solo cuando los has recibido:
         if (Serial.available() > 0)
            {   // lee el byte que ha llegado:
                incomingByte = Serial.read();
                // muestra lo que has recibido:
                Serial.print("He recibido: ");
                Serial.println(incomingByte, DEC);
            }
   }
```

Ejemplo para el Arduino Mega:

```
void setup()
   {   Serial.begin(9600);
       Serial1.begin(9600);
   }

void loop()
   {   // read from port 0, send to port 1:
       if (Serial.available())
          {     int inByte = Serial.read();
                Serial1.print(inByte, BYTE);
          }
       // read from port 1, send to port 0:
       if (Serial1.available())
          {
              int inByte = Serial1.read();
              Serial.print(inByte, BYTE);
          }
   }
```

**Ver también**:

```
-begin()
-end()
-available()
-read()
-peek()
-flush()
-print()
-println()
-write()
```

> *-SerialEvent()*
> *-Stream.available()*

## CATEGORIAS DE LOS PRODUCTOS

Selecciona una categoría