MANUAL LENGUAJE DE PROGRAMACIÓN EN EL ENTORNO ARDUINO

1. ESTRUCTURA

- setup() *(inicialización)*
- loop() *(bucle)*

Estructuras de control

- if *(comparador si-entonces)*
- if...else *(comparador si...sino)*
- for *(bucle con contador)*
- switch case *(comparador múltiple)*
- while *(bucle por comparación booleana)*
- do... while *(bucle por comparación booleana)*
- break *(salida de bloque de código)*
- continue *(continuación en bloque de código)*
- return *(devuelve valor a programa)*

Sintaxis

- ; (punto y coma)
- {} (llaves)
- // (comentarios en una línea)
- /* */ (comentarios en múltiples líneas)

Operadores Aritméticos

- = (asignación)
- + (suma)
- - (resta)
- * (multiplicación)
- / (división)
- % (resto)

Operadores Comparativos

- == (igual a)
- != (distinto de)
- < (menor que)
- > (mayor que)
- <= (menor o igual que)
- >= (mayor o igual que)

Operadores Booleanos

- && (y)
- || (o)
- ! (negación)

Operadores de Composición

- ++ (incrementa)
- -- (decremento)
- += (composición suma)
- -= (composición resta)
- *= (composición multiplicación)
- /= (composición división)

2. VARIABLES

Constantes

- HIGH | LOW
- INPUT | OUTPUT
- true | false
- Constantes Numéricas

Tipos de Datos

- boolean *(booleano)*
- char *(carácter)*
- byte
- int *(entero)*
- unsigned int *(entero sin signo)*
- long *(entero 32b)*
- unsigned long *(entero 32b sin signo)*
- float *(en coma flotante)*
- double *(en coma flotante de 32b)*
- string *(cadena de caracteres)*
- array *(cadena)*
- void *(vacío)*

3. FUNCIONES

E/S Digitales

- pinMode()
- digitalWrite()
- digitalRead()

E/S Analógicas

- analogRead()
- analogWrite() - *PWM (modulación por ancho de pulso)*

E/S Avanzadas

- tone()
- noTone()
- shiftOut()
- pulseIn()

Tiempo

- millis()
- micros()
- delay()
- delayMicroseconds()

Matemáticas

- min() *(mínimo)*
- max() *(máximo)*
- abs() *(valor absoluto)*
- constrain() *(limita)*
- map() *(cambia valor de rango)*
- pow() *(eleva a un número)*
- sq() *(eleva al cuadrado)*
- sqrt() *(raíz cuadrada)*

Trigonometría

- sin() *(seno)*
- cos() *(coseno)*
- tan() *(tangente)*

Números Aleatorios

- randomSeed()
- random()

Comunicación

- Serial

  Funciones

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

1. ESTRUCTURA:

SETUP()

La función setup() se establece cuando se inicia un programa -sketch. Se emplea para iniciar variables, establecer el estado de los pins, inicializar librerías, etc. Esta función se ejecutará una única vez después de que se conecte la placa Arduino a la fuente de alimentación, o cuando se pulse el botón de reinicio de la placa.

Ejemplo

```
int buttonPin = 3;

void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

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

LOOP()

Luego de crear la función setup(), la cual inicializa y prepara los valores iniciales, la función loop() hace justamente lo que su nombre sugiere, por lo tanto se ejecuta consecutivamente, permitiéndole al programa variar y responder. Úsala para controlar de forma activa la placa Arduino.

Ejemplo

```
int buttonPin = 3;

// setup inicializa la comunicación serial y el buttonPin
void setup()
{
  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}

// loop obtiene el estado del pin del botón cada vez,
// y de estar presionado, lo comunica por serial.
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');

  delay(1000);
}
```

ESTRUCTURAS DE CONTROL

IF (CONDICIONAL) Y ==, !=, <, > (OPERADORES DE COMPARACIÓN)

`if`, el cual puede ser usado en conjunto con uno o más operadores de comparación, comprueba si cierta condición se cumple, por ejemplo, si un *input* posee un valor mayor a cierto número. El formato para una comprobación *if* es el siguiente:

```
if (algunaVariable > 50)
{
  // hacer algo aquí.
}
```

Este programa comprueba si la variable *algunaVariable* es mayor a *50*. Si lo es, el programa toma una acción particular. Dicho de otra forma, si la declaración escrita dentro de los paréntesis es verdadera (*true*), el código dentro de las llaves se ejecutará. Sino, el programa ignora dicho código.

Las llaves pueden ser omitidas luego de una declaración *if*. De hacer esto, la siguiente línea (definida por el punto y coma) será la única afectada por la condición.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

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

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

// todos los ejemplos son correctos.
```

Las declaraciones a evaluar dentro de los paréntesis, requieren el uso de uno o más operadores:

Operadores de Comparación:

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

Atención:

Ten cuidado de no usar un signo de igual solo (ej. `if (x = 10)`). Un signo de igual solo es el operador que indica la asignación de un valor, y va a asignar *10* a *x*. En su lugar usa el signo de igual doble (ej. `if (x == 10)`), el cual es el operador de comparación, y compryeba *si* x equivale a 10 o no. El último ejemplo sólo da *true* si x equivale a 10, pero el ejemplo anterior (con un sólo símbolo =) dará siempre *TRUE*.

Esto es porque C evalúa la declaración `if (x=10)` de la siguiente manera: 10 es asignado a x (Recuerda que un signo = solo, es el operador de asignación), por lo tanto x ahora contiene 10. Entonces el condicional *if* evalúa 10, el cual siempre resulta *TRUE*, debido a que cualquier valor numérico mayor a 0 es evaluado como *TRUE*. Consecuentemente, `if (x = 10)` siempre será evaluado como *TRUE*, lo cual no es el resultado deseado cuando se usa una declaración *if*. Adicionalmente, la variable x será definida en 10, lo que tampoco es una acción deseada.

if también puede ser parte de una estructura de control de ramificación usando la construcción if...else.

IF / ELSE

if/else permite mayor control sobre el flujo del código que la declaración *if* básica, por permitir agrupar múltiples comprobaciones. Por ejemplo, un *input* análogo podría ser comprobado, y tomarse una acción si el valor del *input*es menor a 500, y, otra acción se ser igual o mayor a 500. El código se vería así:

```
if (pinCincoInput < 500)
{
  // acción A
}
else
{
  // acción B
}
```

else puede proceder a una comprobación **if**, de esta forma, se pueden realizar múltiples comprobaciones en una misma estructura de condiciones. Cada comprobación procederá a la siguiente, sólo cuando su propio resultado sea *FALSE*. Cuando el resultado sea *TRUE*, su bloque de código contenido, será ejecutado, y el programa esquivará las siguientes comprobaciones hasta el final de la estructura de comprobaciones. Si ninguna comprobación devuelve valor *TRUE*, el **else** será ejecutado, y de no haber ninguno declarado, simplemente no sucede nada.

Entonces un bloque **else if** puede ser usado con o sin **else** al final. La cantidad de declaraciones **else if**, y sus ramificaciones son ilimitadas.

```
if (pinCincoInput < 500)
{
  // ejecutar A
}
else if (pinCincoInput >= 1000)
{
  // ejecutar B
}
else
{
  // ejecutar C
}
```

Otra forma de expresar ramificaciones (*branching* en inglés), y realizar comprobaciones mutuamente exclusivas, es con la declaración switch case, la cual resulta más idónea en algunos casos.

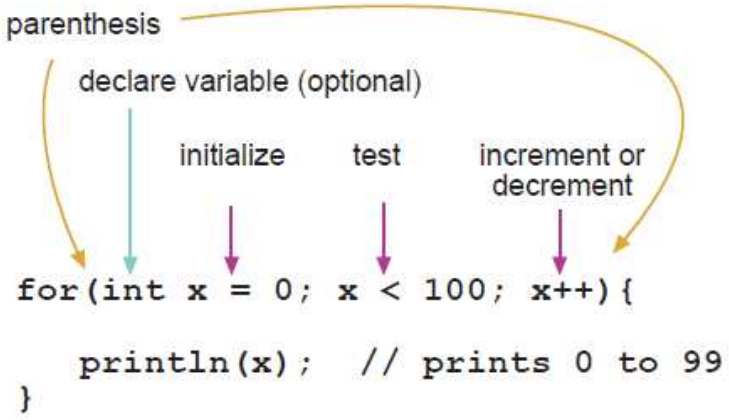
DECLARACION FOR

Descripción

La declaración **for** es usada para repetir un bloque encerrado entre llaves. Un incremento de un contador es usado, normalmente, para aumentar y terminar con el bucle. La estructura **for** es muy útil para la mayoría de las operaciones repetitivas, y habitualmente se usa para operaciones con vectores, para operar sobre conjuntos de datos/pines

El bucle **for** tiene tres partes o argumentos en su inicialización:

```
for (initialization; condition; increment) {
//función(es);
}
```



La **initialization**, o inicialización, se produce sólo la primera vez. Cada vez que se va a repetir el bucle, se revisa la **condition**, o condición: si es cierta, el bloque de funciones (y el incremento del contador) se ejecutan, y la condición vuelve a ser comprobada de nuevo. Si la condición es falsa, el bucle termina.

Ejemplo

```
// Variar la intensidad de un LED usando un salida PWM
int PWMpin = 10; // En el pin 10 hay un LED en serie con una
resistencia de 470 ohmios

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

void loop()
{
   for (int i=0; i <= 255; i++){
      analogWrite(PWMpin, i);
      delay(10);
   }
}
```

Consejos de programación

El bucle **for**, en C, es mucho más flexible que otros bucles **for** en otros lenguajes, incluyendo BASIC. Cualquiera (o todos) los parámetros pueden ser omitidos, sin embargo los puntos y coma (;) son obligatorios. También las inicialización, condición e incremento pueden ser cualquier declaración en C válida, con variables independientes, y podemos usar cualquier tipo de variable, incluídos los float. Estos tipos de declaración **for** poco usuales pueden proporcionar una solución válida a algunos problemas de programación raros.

Por ejemplo, usando la multiplicación en el parámetro de incremento, podemos generar una progresión logarítmica.

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

Este código generará: 2,3,4,6,9,13,19,28,42,63,94

Otro ejemplo es hacer apagarse/encenderse un LED poco a poco, Another example, fade an LED up and down with one **for** loop:

```
void loop()
{
   int x = 1;
   for (int i = 0; i > -1; i = i + x){
      analogWrite(PWMpin, i);
      if (i = 255) x = -1;            // cambia de signo para apagarlo
      delay(10);
   }
}
```

SENTENCIA SWITCH / CASE

Como las sentencias **if**, **switch...case** controla el flujo de programas permitiendo a los programadores especificar diferentes códigos que deberían ser ejecutados en función de varias condiciones. En particular, una sentencia switch compara el valor de una variable con el valor especificado en las sentencias case. Cuando se encuentra una sentencia case cuyo valor coincide con dicha variable, el código de esa sentencia se ejecuta.

La palabra clave **break** sale de la sentencia switch, y es usada típicamente al final de cada case. Si una sentencia break, la sentencia switch continuaría ejecuntándo las siguientes expresiones ("falling-through") hasta encontrar un break, o hasta llegar al final de la sentencia switch.

Ejemplo

```
switch (var) {
  case 1:
    //hacer algo cuando sea igual a 1
    break;
  case 2:
    //hacer algo cuando sea igual a 2
    break;
  default:
    // si nada coincide, ejecuta el "default"
    // el "default" es opcional
}
```

Sintaxis

```
switch (var) {
  case etiqueta:
    // sentencias
    break;
  case etiqueta:
    // sentencias
    break;
  default:
    // sentencias
}
```

Parámetros

var: la variable cuyo valor comparas con los varios "case"

etiqueta: un valor para comparar con la variable

BUCLES WHILE

Descripción

Los bucles **while** se ejecutan continuamente, hasta que la expresión de dentro del paréntesis, (), pasa a ser falsa. Algo debe modificar la variable comprobada, el bucle **while** nunca terminará. Lo que modifique la variable puede estar en el código, como una variable que se incrementa, o ser una condición externa, como el valor que da un sensor.

Sintaxis

```
while(expresion){
  // sentencia(s)
}
```

Parámetros

expresion - una sentencia C (booleana) que da como valor verdadero (true) o falso (false)

Ejemplo

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

DO - WHILE

El bucle "'do'" trabaja de la misma manera que el bucle "'while'", con la excepción de que la condición se comprueba al final del bucle, por lo que este bucle se ejecuta "siempre" al menos una vez.

```
do
{
    // bloque de instrucciones
} while (condición);
```

Ejemplo:

```
do
{
  delay(50);              // espera a que los sensores se estabilicen
  x = readSensors();  // comprueba los sensores

} while (x < 100);    //si se cumple la condición se repite el bucle
```

BREAK

break es usado para salir de los bucles **do**, **for**, o **while**, pasando por alto la condición normal del bucle. Es usado también para salir de una estructura de control **switch**.

Ejemplo

```
for (x = 0; x < 255; x ++)
{
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      // bail out on sensor detect
       x = 0;
       break;   // sale del bucle for.
    }
    delay(50);
}
```

CONTINUE

La sentencia continue omite el resto de iteraciones de un bucle (**do**, **for**, o **while**). Continúa saltando a la condición de bucle y procediendo con la siguiente iteración.

Ejemplo

```
for (x = 0; x < 255; x ++)
{
    if (x > 40 && x < 120){      // crea un salto en estos valores
        continue;
    }

    digitalWrite(PWMpin, x);
    delay(50);
}
```

RETURN

Termina una función y devuelve un valor a la función que la llama. Puede no devolver nada.

Sintaxis

return;

return valor; // ambas formas son correctas

Parámetros

valor: cualquier variable o tipo constante

Ejemplos

Una función que compara la entrada de un sensor a un umbral

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

La palabra clave *return* es útil para depurar una sección de código sin tener que comentar una gran cantidad de líneas de código posiblemente incorrecto.

```
void loop(){

// código magnífico a comprobar aquí

return;

// el resto del programa del que se desconfía
// que nunca será ejecutado por estar detrás de return
}
```

SINTAXIS

; PUNTO Y COMA

Utilizado para terminar una declaración.

Ejemplo

```
int a = 13;
```

Truco

Olvidarse de terminar una línea con el punto y coma hará que se lance un error de compilación. El texto del error puede ser obvio y referirse concretamente a la ausencia del punto y coma, pero puede no hacerlo. Si aparece un error de compilación impenetrable y aparentemente incoherente, una de las primeras cosas a hacer es comprobar que no falta ningún punto y coma en la vecindad de la línea en la que el compilador da el error.

{} LLAVES

Las Llaves son un parte importante del lenguaje de programación C. Se utilizan en diferentes construcciones (ver ejemplos al final), esto a veces puede ser confuso para los principiantes.

Una llave de apertura "{" siempre debe ir seguida de una llave de cierre "}". Esta es una condición a la que se suele referir como llaves emparejadas. El IDE (Entorno Integrado de Desarrollo) Arduino incluye una característica para comprobar si las llaves están emparejadas. Sólo tienes que seleccionar una Llave o incluso hacer click en el punto de inserción que sigue inmediatamente a una llave, y su compañera lógica será seleccionada.

En la actualidad esta característica tiene un pequeño fallo, el IDE encuentra a menudo (incorrectamente), llaves en el texto que pueden estar situadas dentro de comentarios.

Los programadores principiantes y los programadores que llegan a C desde el lenguaje BASIC a menudo encuentran dificultades o grandes confusiones usando llaves. Después de todo, las llaves reemplazan el RETURN en una subrutina(función), el ENDIF en un condicional y el NEXT en un loop FOR.

Dado que el uso de las llaves es tan variado, es una buena práctica de programación escribir la llave de cierre inmediatamente después de escribir la llave de apertura, cuando se inserta una construcción que requiere llaves. Después insertar algunos saltos de línea (líneas en blanco) entre las llaves y comenzar a insertar sentencias. De esta forma tus llaves y su posición, nunca llegarán a estar desemparejadas.

Llaves desemparejadas a menudo pueden conducir a errores de compilación misteriosos y difíciles de comprender, que pueden ser complicados de rastrear en un programa largo. Debido a sus variados usos, las llaves también son increíblemente importantes para la sintaxis de un programa, el movimiento de una llave una o dos líneas, a menudo afecta de manera dramática el significado de un programa.

Usos principales de las Llaves

Funciones

```
void myfunction(tipodato argumento){
  sentencia(s)
}
```

Loops

```
while (expresión booleana)
{
  sentencia(s)
}

do
{
```

```
   sentencia(s)
} while (expresión booleana);

for (inicialización; condición final; expresión incremento)
{
  sentencia(s)
}
```

Sentencias Condicionales

```
if (expresión booleana)
{
   sentencia(s)
}

else if (expresión booleana)
{
   sentencia(s)
}
else
{
   sentencia(s)
}
```

// COMENTARIOS

Los comentarios son líneas en el programa para aclarar a tí o a otros sobre el funcionamiento del programa. Estas líneas son ignoradas por el compilador y no se exportan al procesador. No ocupan por tanto espacio en el Chip Atmega

El único propósito de los comentarios es que entiendas o entiendan (o recuerdes) cual es el funcionamiento de tu programa. Existen dos maneras distintas de marcar una línea como comentario:

Ejemplo

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

/* Esto es un comentario multilínea - úsalo para comentar bloques
enteros de código

if (gwb == 0){   // Una línea de comentario sencilla puede usarse dentro
                  de un comentario multilínea
x = 3;               /* pero no otro comentario multilínea- esto no es
válido */
}
// No olvides cerrar el comentario multilínea
*/
```

Consejo
Cuando estás experimentando con código, "comentar" partes del programa es una útil de eliminar líneas que puedan contener errores. Así dejamos las líneas en el código, pero como comentario , así que serán ignoradas por el compilador. Es especialmente útil cuando estamos intentando localizar el problema , o cuando el compilador rechaza el programa.

/**/ COMENTARIOS

Los comentarios son líneas en el programa para aclarar a tí o a otros sobre el funcionamiento del programa. Estas líneas son ignoradas por el compilador y no se exportan al procesador. No ocupan por tanto espacio en el Chip Atmega

El único propósito de los comentarios es que entiendas o entiendan (o recuerdes) cual es el funcionamiento de tu programa. Existen dos maneras distintas de marcar una línea como comentario:

Ejemplo

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

/*  Esto es un comentario multilínea - úsalo para comentar bloques
enteros de código

if (gwb == 0){   // Una línea de comentario sencilla puede usarse dentro
                  de un comentario multilínea
x = 3;       /* pero no otro comentario multilínea- esto no es válido */
}
// No olvides cerrar el comentario multilínea
*/
```

Consejo
Cuando estás experimentando con código, "comentar" partes del programa es una útil de eliminar líneas que puedan contener errores. Así dejamos las líneas en el código, pero como comentario , así que serán ignoradas por el compilador. Es especialmente útil cuando estamos intentando localizar el problema , o cuando el compilador rechaza el programa.

OPERADORES ARITMÉTICOS

= OPERADOR DE ASIGNACIÓN (UN SOLO SÍMBOLO DE "IGUAL")

Guarda el valor en la derecha del símbolo "=" dentro de la variable a la izquierda del símbolo "=".

El signo de igualdad "=" en el lenguaje de programación C se llama el operador de asignación. Tiene un significado diferente que en la clase de álgebra en el que se indica una ecuación o igualdad. El operador de asignación le dice al microcontrolador que evalúe cualquier valor o expresión en el lado derecho del signo igual, y lo almacene en la variable a la izquierda del signo igual.

Ejemplo

```
int sensVal;                // declara una variable int llamada sensVal
senVal = analogRead(0);   // guarda el valor (digitalizado) del   voltaje
                           de entrada del pin analógico 0 en SensVal
```

Sugerencias de programación

The variable on the left side of the assignment operator (= sign) needs to be able to hold the value stored in it. If it is not large enough to hold a value, the value stored in the variable will be incorrect.

Don't confuse the assignment operator [=] (single equal sign) with the comparison operator [==] (double equal signs), which evaluates whether two expressions are equal.

La variable en el lado izquierdo del operador de asignación (signo "=") tiene que ser capaz de mantener el valor almacenado en ella. Si no es suficientemente grande para contenerlo, el valor almacenado en la variable será incorrecto.

No confunda el operador de asignación [=] (un solo signo igual) con el operador de comparación [==] (un signo

igual doble), que evalúa si dos expresiones son iguales.

SUMA, RESTA, MULTIPLICACIÓN Y DIVISIÓN

Descripción

Estos operadores devuelven la suma, diferencia, producto o cociente (respectivamente) de los dos operandos. La operación se lleva a cabo utilizando el tipo de datos de los operandos, por lo que, por ejemplo, `9 / 4` resulta `2` desde 9 y 4 que son enteros int. Esto también significa que la operación puede desbordarse si el resultado es mayor que el que se puede almacenar en el tipo de datos (por ejemplo, la suma de 1 a un int con el valor de 32.767 resulta -32.768). Si los operandos son de tipos diferentes, se utiliza el tipo del "más grande" para el cálculo.

Si uno de los números (operandos) es del tipo **float** o del tipo **double**, se usará coma flotante para el cálculo.

Ejemplos

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

Sintaxis

```
result = value1 + value2;
result = value1 - value2;
result = value1 * value2;
result = value1 / value2;
```

Parámetros:

value1: cualquier variable o constante

value2: cualquier variable o constante

Sugerencias de programación:

- Debes saber que las integer constants por defecto son int, así que algunos cálculos con constantes pueden provocar desbordamiento (p.e. 60 * 1000 devolverá un resultado negativo)
- Elige los tamaños de variable que sean suficientemente grandes como para alojar el resultado de tus calculos.
- Debes saber en que punto tu variable se desbordará en su máximo, y que esto también ocurre en su mínimo. p.e. (0 - 1) o también (0 - - 32768)
- Para cálculos matemáticos que requieren fracciones, usa variables float, pero ten en cuenta los inconvenientes: gran tamaño, velocidades bajas de cálculo
- Usa el operador de conversión (casting). Por ejemplo: (int)myFloat para convertir el tipo de una variable en el momento.

% (MÓDULO)

Descrición

Calcula el resto de la división entre dos enteros. Es útil para mantener una variable dentro de un rango particular (por ejemplo el tamaño de un array)

Sintaxis

resultado = dividendo % divisor

Parametros

dividendo: el número que se va a dividir

divisor: el número por el que se va a dividir

Devuelve

el resto de la división

Ejemplo

```
x = 7 % 5;   // x ahora contiene 2
x = 9 % 5;   // x ahora contiene 4
x = 5 % 5;   // x ahora contiene 0
x = 4 % 5;   // x ahora contiene 4
```

Código de ejemplo

```
/* actualiza un valor en el array cada vez que se pasa por el bucle */

int valores[10];
int i = 0;

void setup() {}

void loop()
{
  valores[i] = analogRead(0);
  i = (i + 1) % 10;   // el operador módulo prevalece sobre la variable
}
```

Nota

El operador modulo no funciona con datos en coma flotante (float)

OPERADORES BOOLEANOS

Se pueden usar dentro de operaciones condicionales o en una sentencia if.

&& (AND lógico)

Verdadero sólo si ambos operadores son Verdadero, por ejemplo:

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

Es Verdadero sólo si ambas entradas estás activadas, es decir, en estado HIGH.

|| (OR lógico)

Verdadero si alguno de los dos operadores es Verdadero, por ejemplo:

```
if (x > 0 || y > 0) {
  // ...
}
```

Es Verdadero si alguno de los valores x ó y es mayor que 0.

! (NOT)

Verdadero si el operador es Falso, por ejempo:

```
if (!x) {
  // ...
}
```

Es Verdadero si el valor de x es Falso (p. e. si x es igual a 0).

Atención.

Asegúrate de no confundir el operador AND booleano && (doble signo &) con el operador AND para bits & (un solo signo &).

De la misma manera no hay que confundir el operador OR booleano || (doble barra vertical) con el operador OR para bits | (una sola barra vertical).

El operador NOT de bits ~ (tilde) tiene una apariencia muy diferente del operador NOT booleano ! (exclamación o "bang", como lo llaman algunos programadores), pero debes asegurarte de usar cada uno de ellos dentro del contexto adecuado.

Ejemplos.

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

OPERADORES DE COMPOSICIÓN

++ (INCREMENTO) / -- (DISMINUCIÓN)

Descripción

Incrementa o disminuye una variable

Sintaxis

```
x++;  // incrementa x en uno y regresa el valor anterior de x
++x;  // incrementa x en uno y regresa el nuevo valor de x

x-- ;   // disminuye x en uno y y regresa el valor anterior de x
--x ;   // disminuye n uno y regresa el nuevo valor de x
```

Parámetros

x: un entero o long (puede aceptar sin signo)

Devuelve

El valor original o el resultante del incremento o disminución de la variable.

Ejemplos

```
x = 2;
y = ++x;      // x ahora guarda 3, y guarda 3
y = x--;      // x guarda 2 de nuevo, y sigue guardando 3
```

+= , -= , *= , /=

Descripción

Realiza una operación matemática con una variables con respecto a otra variable o una constante. El operador += (y los demás operadores) son una forma simplificada de la sintaxis completa, tal y como se muestra más abajo.

Sintaxis

```
x += y;   // equivalente a la expresión x = x + y;
x -= y;   // equivalente a la expresión x = x - y;
x *= y;   // equivalente a la expresión x = x * y;
x /= y;   // equivalente a la expresión x = x / y;
```

Parámetros

x: cualquier tipo de variable

y: cualquier tipo de variable o constante

Ejemplos

```
x = 2;
x += 4;      // x ahora es 6
x -= 3;      // x ahora es 3
x *= 10;     // x ahora es 30
x /= 2;      // x ahora es 15
```

2. VARIABLES:

CONSTANTES

Las constantes variables que vienen predefinidas en el lenguaje de Arduino. Se usan para facilitar la lectura de los programas. Clasificamos las constantes en grupos.

Las que definen niveles lógicos, verdadero (true) y falso (false) (Constantes Booleanas)

Existen dos constantes para representar si algo es cierto o falso en Arduino: **true**, y **false**.

FALSE

false es el más sencillo de definir. false se define como 0 (cero).

TRUE

true se define la mayoría de las veces como 1, lo cual es cierto, pero tiene una definición más amplia. Cualquier entero que es *no-cero* es TRUE, en un sentido Booleano. Así, en un sentido Booleano, -1, 2 y -200 son todos true.

Ten en cuenta que las constantes *true* y *false* se escriben en minúsculas, al contrario que HIGH, LOW, INPUT, y OUTPUT.

Las que definen el nivel de los pines, nivel alto (HIGH) y nivel bajo (LOW)

Cuando leemos o escribimos en un pin digital, existen sólo dos valores que podemos obtener o asignar : **HIGH** y **LOW**.

HIGH

El significado de HIGH (en referencia a un pin) depende de si el pin está configurado como entrada (INPUT) o como salida (OUTPUT). Cuando un pin se configura como entrada (INPUT) usando pinMode, y se lee con digitalRead, el microcontrolador nos retornará HIGH si en el pin hay 3 voltios o más.

Un pin puede ser configurado como entrada (INPUT) usando pinMode, y después establecerlo a HIGH con digitalWrite, esto conectará el pin a 5 Voltios a través de una resistencia interna de 20K, resistencia pull-up , la cual establecerá el pin al estado de lectura HIGH a menos que la conectemos a una señal LOW a través de un circuito externo.

Cuando un pin se configura como salida (OUTPUT) con pinMode, y se establece a HIGH con digitalWrite, el pin tiene 5V. Ene ste estado puede usarse como *fuente* de corriente , e.j. Luz y LED que se conectan a través de resistencias en serie a masa (tierra), o a otro pin configurado como salida y establecido a LOW.

LOW

El significado de LOW difiere también según esté configurado como entrada (INPUT) o como salida (OUTPUT). Cuando un pin está configurado como entrada (INPUT) con pinMode, y se lee con digitalRead, el microcontrolador retornará LOW si el voltaje presente en el pin es de 2V o menor.

Cuando un pin es configurado como salida (OUTPUT) con pinMode,y establecido LOW con digitalWrite, el pin tiene 0 voltios. En este estado puede *meter* corriente, e.j. Luz y LED que se conectan a través de resistencias en serie a +5 voltios, o a otro pin configurado como salida, y establecido a HIGH.

Las que definen los pines digitales, INPUT y OUTPUT

Los pines digitales pueden se usados como **entrada (INPUT)** o como **salida (OUTPUT)**. Cambiando un pin de INPUT a OUTPUT con pinMode() el comportamiento eléctrico del pin cambia drásticamente.

Pins configurados como entradas

Los pins de Arduino (Atmega) configurados como **INPUT** con pinMode() se dice que se encuentran en un estado de alta impedancia. Una forma de explicar esto es que un pin configurado como entrada se le aplica una muy baja demanda, es decir una resistencia en serie de 100 Megohms. Esto lo hace muy útil para leer un sensor, pero no para alimentar un LED.

Pins configurados como salidas

Los pins configurados como **salida (OUTPUT)** con pinMode() se dice que estan en estado de baja impedancia. Esto implica que pueden proporcionar una sustancial cantidad de corriente a otros circuitos. Los pins de Atmega pueden alimentar (proveer de corriente positiva) or meter (proveer de masa) hasta 40 mA (miliamperios) de corriente a otros dispositivos/circuitos. Esto o hace muy útil para alimentar LED's pero inservible para leer sensores. Los pins configurados como salida pueden deteriorarse o romperse si ocurre un cortocircuito hacia los 5V o 0V . La cantidad de corriente que puede proveer un pin del Atmega no es suficiente para la mayoría de los relés o motores, y es necesario añadir circuitería extra.

INTEGER CONSTANTS

Integer Constants son números utilizados directamente en un *sketch*, como `123`. Por defecto, éstos números son tratados como int, pero puedes cambiarlo con las letras *U* y *L* (ver abajo).

Normalmente, las constantes *integer* son tratadas como enteros base 10 (decimales), pero se puede utilizar notación especial (formateadores) para ingresar números en otras bases.

Base	Ejemplo	Formateador	Comentario
10 (decimal)	123	Ninguno.	
2 (binario)	B1111011	Antecede "B"	Sólo funciona con valores de 8 bits (0 to 255). Caracteres válidos 0-1
8 (octal)	0173	Antecede "0"	Caracteres válidos 0-7
16 (hexadecimal)	0x7B	Antecede "0x"	Caracteres 0-9, A-F, a-f válidos.

Decimal es base 10. Esta es la matemática de sentido común con que se conocen. Para constantes sin otros prefijos, se asume el formato decimal.

Ejemplo:

```
101     // igual a 101 decimal   ((1 * 10^2) + (0 * 10^1) + 1)
```

Binary es base dos. Sólo caracteres 0 y 1 son válidos.

Ejemplo:

```
B101    // igual a 5 decimal   ((1 * 2^2) + (0 * 2^1) + 1)
```

El formateador binario sólo funciona en *bytes* (8 *bits*) entre 0 (B0) y 255 (B11111111). Si resulta conveniente ingresar un entero (*int*, 16 *bits*) de forma binaria, puedes hacer un procedimiento de dos pasos, como a continuación:

```
valorInt = (B11001100 * 256) + B10101010;    // B11001100 es el ''byte'' alto.
```

Octal es base ocho. Sólo caracteres de 0 hasta 7 son válidos. Los valores Octales son indicados por el prefijo "0"

Ejemplo:

```
0101    // igual a 65 decimal   ((1 * 8^2) + (0 * 8^1) + 1)
```

Atención
Es posible generar un *bug* o error difícil de encontrar al incluír (de forma involuntaria) un cero antecediento una constante, logrando así que el compilador interprete tu constante como octal.

Hexadecimal (ó hex) es base dieciséis. Los caracteres válidos son del 0 al 9, y las letras desde la *A* hasta la *F*; *A* tiene el valor de 10, *B* es 11, *C* es 12, *D* es 13, *E* es 14, y la F, como ya habrás adivinado, es 15. Los valores hexadecimales se indican con el prefijo "0x". Nota que los valores de la *A* a la *F*, pueden ser escritos en mayúscula o minúscula.

Ejemplo:

```
0x101   // igual a 257 decimal   ((1 * 16^2) + (0 * 16^1) + 1)
```

TIPOS DE DATOS

BOOLEANOS

Un **booleano** sólo puede tomar dos valores, Verdadero o Falso. Cada booleano ocupa un único byte en la memoria.

Ejemplo:

```
int LEDpin = 5;       // LED en el pin 5
int switchPin = 13;   // pulsador en el pin 13, su otra patilla conectada en GND

boolean running = false;  // crea la variable booleana running y le asisga el
valor Falso (false)

void setup()
{
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);      // conecta la resistencia pull-up interna
del pin 13
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // si el pulsador es accionado la pull-up mantiene el pin en estado HIGH
    delay(100);                         // retardo para impedir un rebote en el
pulsador
    running = !running;                    // invierte el valor de la variable
running
    digitalWrite(LEDpin, running)      // enciende el LED
  }
}
```

CHAR

Descripción

Es un tipo de dato que ocupa un byte de memoria y almacena un valor de carácter. Los carácteres literales se escriben con comillas simples: 'A' (para varios carácteres -strings- utiliza dobles comillas "ABC").

De todas maneras los carácteres son almacenados como números. Puedes ver su codificado en la tabla ASCII. Con esto podemos entender que es posible realizar cálculos aritméticos con los carácteres, en este caso se utiliza el valor ASCII del carácter (por ejemplo 'A' + 1 tiene el valor de 66, ya que el valor ASCII de la letra mayúscula A es 65). Mira Serial.println para tener mas información de como son traducidos a números los carácteres.

El tipo de datos char tiene signo. esto significa que codifica números desde -128 hasta 127. Para un dato de un byte (8bits), utiliza el tipo de dato "byte".

Ejemplo

```
  char miChar = 'A';
  char miChar = 65;      // los dos son equivalentes
```

BYTE

Descripción

Un byte almacena un número sin signo de 8-bit, desde 0 hasta 255.

Ejemplo

```
byte b = B10010;  // "B" es el formateador binario (B10010 = 18 decimal)
```

INT

Descripción

Integers (Números enteros) son el principal tipo de datos para almacenar números, y guardan valores de **2 bytes**. Esto produce un rango entre *-32,768* hasta *32,767* (valor mínimo de -2^15 y un valor máximo de (2^15) - 1).

Variables tipo **Int**, almacenan números negativos con una técnica llamada Complemento a dos. El *bit* más alto, a veces llamado como *"sign" bit*, indica que el número es negativo. Se invierte el valor de cada uno de los *bits*, es decir, se realiza el complemento a uno, y se suma 1 al número obtenido.

La placa Arduino, se encarga de tratar con números negativos por tí, para que las operaciones aritméticas trabajen de manera transparente y en la forma esperada. Sin embargo, pueden haber complicaciones inesperadas al tratar con el operador right bitshift (>>).

Ejemplo

```
int ledPin = 13;
```

Syntax

```
int var = val;
```

- var - nombre de la variable *int*.
- val - valor asignado a dicha variable.

Consejo

Cuando las variables son hechas para exceder su límite, éstas vuelven a su capacidad mínima, ésto sucede en ambas direcciones:

```
   int x
   x = -32,768;
   x = x - 1;        // x ahora contiene 32,767 - vuelve a empezar en dirección
contraria.

   x = 32,767;
   x = x + 1;       // x ahora contiene -32,768 - vuelve a empezar.
```

UNSIGNED INT

Descripción

Los enteros sin firma (unsigned int) son los mismos enteros de modo que almacenan un valor de dos bytes. En lugar de almacenar números negativos, sólo almacenan valores positivos, generando un rango útil desde 0 a 65,535 (2^16 - 1).

La diferencia entre los enteros sin firma y los enteros (firmados), reside en que el bit más alto, a veces referenciado como el bit "firmado", es interpretado. En el tipo de datod int de Arduino (que es firmado), si el bit más alto es "1", el número es interpretado como un número negtivo, y los otros 15 bits son interpretados con complemento a 2.

Ejemplo

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

Sintáxis

```
unsigned int var = val;
```

- var - el nombre de tu variable unsigned int
- val - el valor que asignas a a esa variable

Consejo de codificación

Cuando las variables sobrepasan su capacidad máxima dan la vuelta a su mínima capacidad. Ten en cuenta que esto sucede en ambas direcciones.

```
unsigned int x
x = 0;
x = x - 1;         // x ahora contiene 65535 - da la vuelta en dirección negativa
x = x + 1;       // x ahora contiene 0 - da la vuelta
```

LONG

Descripción

Las variables de tipo Long son variables de tamaño extendido para almacenamiento de números, y 32 bits (4 bytes), desde -2,147,483,648 hasta 2,147,483,647.

Ejemplo

```
long speedOfLight = 186000L;   // ver Constantes Integer para la explicación de la 'L'
```

Sintaxis

```
long var = val;
```

- var - nombre de la variable tipo Long
- val - valor asignado a la variable

UNSIGNED LONG

Descripción

Las variable long sin firmar (unsigned long) son variables extendidas para almacenar números, y almacenar 32 bits (4 bytes). Por el contrario que las variables long estándar, las unsigned long no almacenan números negativos, haciendo que su rango sea de 0 a 4,294,967,295 (2^32 - 1).

Ejemplo

```
unsigned long tiempo;

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

void loop()
{
  Serial.print("Tiempo: ");
  tiempo = millis();
  //imprime el tiempo desde el inicio del programa
```

```
  Serial.println(tiempo);
  // espera un segundo para no enviar cantida de masivas de datos
  delay(1000);
}
```

Sintáxis

```
unsigned long var = val;
```

- var - el nombre de tu variable long
- val - el valor que asignas a tu variable long

FLOAT

Descripcion

El tipo variable para los números en coma flotante (número decimal). Estos números son usados, habitualmente, para aproximar valores analógicos y continuos, debido a que ofrecen una mayor resolución que los enteros. Las variables tipo **float** tienen el valor máximo 3.4028235E+38, y como mínimo pueden alcanzar el -3.4028235E+38. Ocupan 4bytes (32bits).

Los **floats** tienen una precisión de 6 o 7 dígitos decimales. Esto significa el número total de dígitos, no el número a la derecha de la coma decimal. Al contrario que en otras plataformas, donde tu podrías obtener mayor precisión usando una variable tipo double (por ejemplo, por encima de 15 dígitos), en Arduino los double tienen el mismo tamaño que los float.

Los números en coma flotante no son exactos, y muchos proporcionan falsos resultados cuando son comparados. Por ejemplo, `6.0 / 3.0` puede no ser igual a `2.0`. Debes comprobar que el valor absoluto de la diferencia entre los números pertenezca a un rango pequeño.

La matemática en coma flotante es mucho más lenta que la matemática de enteros para realizar operaciones, por lo que deberías evitarla si, por ejemplo, un bucle tiene que ejecutarse a la máxima velocidad para funciones con temporizaciones precisas. Los programadores normalmente suelen asignar unas longitudes para convertir las operaciones de coma flotante en cálculos con enteros, para aumentar la velocidad.

Ejemplos

```
float myfloat;
float sensorCalbrate = 1.117;
```

Sintaxis

```
float var = val;
```

- **var** - el nombre de la variable tipo float
- **val** - el valor que le asignas a esa variable

Código de ejemplo

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

x = 1;
y = x / 2;            // y ahora contiene 0, la parte entera de la operación
z = (float)x / 2.0;   // z ahora contiene 0.5 (se debe usar 2.0, en lugar de
2)
```

DOUBLE

Descripcion

Número en coma flotante de doble precisión. Ocupa 4 bytes.

La implementación "double" en Arduino es exactamente lo mismo que la FLOAT, sin ganar nada de precisión.

Consejo

Los usuarios que porten código de otras fuentes y que incluyan variable tipo double deberían examinar el código para ver si la precisión necesaria es diferente a la que se puede lograr en Arduino.

STRING

Descripción

Los strings se representan como arrays de caracteres (tipo char) que terminan con el caracter NULL.

Ejemplos

Todas las siguientes son declaraciones válidas de strings.

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[ ] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

Posibilidades de declaraciónd e strings

- Declarar un array de caracteres sin incializarlo como en Str1
- Declarar un array de caracteres (con un caracter extra) y el compilador añadirá el caracter NULL requerido, como en STR2
- Explicitar el caracter NULL, Str3
- Inicializar con un string constante entre comillas dobles; el compilador medirá el tamaño del array para ajustar el string constante y caracter NULL para finalizar, Str4
- Inicializar el array con un tamaño explícito y un string constante, Str5
- Inicializar el array, dejando un espacio extra para un string más largo, Str6

Terminación NULL

Generalmente, los strings se finalizan con un caracter NULL (código ASCII 0). Esto permite a funciones (como Serial.print()) establecer dónde está el final del string. De otra forma, seguiría leyendo los siguientes bytes de la memoria que no forman parte del string.

Esto significa que tu string necesita tener espacio para un caracter más del texto que quieres que contenga. Esto es por lo que Str2 y Str5 necesitan 8 caracteres, incluso aunque "arduino" tenga sólo 7 - la última posición es automáticamante completada con un caracter NULL. Str4 será automáticamente dimensionada a 8 caracteres, uno para el NULL extra. En Str3, hemos incluido nosotros mismos el caracter NULL (escrito como '\0').

Ten en cuenta que es posible tener un string sin un caracter NULL al final (p.e. si tú has especificado la longitud de Str2 como 7 en lugar de 8). Esto romperá la mayoría de funciones que usen strings, por lo que no deberías hacerlo intencionadamente. Por lo tanto, si detectas algún comportamiento extraño (operando con los caracteres, no con el string), este podría se el problema

Comillas simples o dobles

Los string siempre se definen entre comillas dobles ("Abc") y los caracteres siempre se definen dentro de comillas

simples ('A').

Envolviendo string largos

Puedes envolver strings largos de ésta manera:

```
char miString[] = "Esta es la primera linea"
" esta es la segunda linea"
" etcetera";
```

Arrays de strings

Amenudo es conveniente, al trabajar con grandes cantidades de texto, como proyectos con displays LCD, configurar un array de strings. Como los strings son en sí mismo arrays, esto es un ejemplo real de un array bidimensional.

En el código de abajo, el asterisco después del tipo de dato char "char*" indica que es un array de "punteros". Todos los nombres de arrays son punteros, por lo que esto es necesario para crear un array de aarys. Los punteros son una de las partes más exotéricas de C como para que los principieantes puedan llegar a entenderlas, pero no es necesario entender punteros en detalle para usarlos con efectividad.

Ejemplo

```
char* miStrings[]={"Este es el string 1", "Este es el string 2", "Este es el
string 3",
"Este es el string 4", "Este es el string 5","Este es el string 6"};

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

void loop(){
for (int i = 0; i < 6; i++){
   Serial.println(miStrings[i]);
   delay(500);
   }
}
```

ARRAYS

Una matriz o "array" es una colección de variables que son accedidas mediante un número de índice. Los "arrays" en el lenguaje de programación C, en el cual está basado Arduino, pueden ser complicadas, pero usar "arrays" simples es relativamente sencillo.

Creando (Declarando) una matriz o Array

Todos los métodos de abajo son formas válidas de crear (declarar) una matriz.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {2, 4, -8, 3, 2};
char message[6] = "hola";
```

Puedes declarar una matriz sin inicializarla como en myInts.

En myPins se declara una matriz sin establecer un tamaño explícito. El compilador cuenta el número de elementos y crea la matriz con el tamaño apropiado.

Finalmente, puedes tanto declarar con un tamaño la matriz e inicializarla al mismo tiempo, como en mySensVals

Accediendo a una matriz o Array

Los Arrays son **zero indexed**, esto significa que, al referirse a una matriz, el primer elemento de la matriz está en el indice 0. Por lo tanto:

`mySensVals[0] == 2, mySensVals[1] == 4,` y sucesivos.

Esto también significa que en una matriz con 10 elementos. el indice 9 es el último elemento. Por lo tanto:

```
int myArray[10]={9,3,2,4,3,2,7,8,9,11};
     // myArray[9]    contiene 11
     // myArray[10]    es invalido y contiene información aleatoria (de otra
dirección de memoria)
```

Por esta razón hay que tener cuidado en el acceso a las matrices. El acceso más allá del final de una matriz (usando un número de índice mayor que el tamaño declarado - 1) resultará la lectura de la memoria que está en uso para otros fines. La lectura de estos lugares probablemente no va a hacer mucho mal, excepto la lectura de datos no válidos. Escribir en las localidades de memoria aleatoria es definitivamente una mala idea y, a menudo puede conducir a resultados inesperados como fallos o mal funcionamiento del programa. Esto también puede ser un error difícil encontrar.

A diferencia de BASIC o JAVA, el compilador de C no realiza ninguna comprobación para ver si el acceso a una matriz está dentro de los límites del tamaño de la matriz que ha declarado.

Para asignar un valor a una matriz:

```
mySensVals[0] = 10;
```

To retrieve a value from an array:

```
x = mySensVals[4];
```

Matrices y los bucles FOR

Arrays are often manipulated inside **for** loops, where the loop counter is used as the index for each array element. For example, to print the elements of an array over the serial port, you could do something like this: Las matrices se utilizan muchas veces en el interior de bucles **for**, donde el contador de bucle se utiliza como el índice de cada elemento de la matriz. Por ejemplo, para imprimir los elementos de una matriz a través del puerto serie, se podría hacer algo como esto:

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

VOID

La palabra reservada void se usa sólo en la declaración de funciones. Indica que se espera que no devuleva información a la función donde fué llamada.

Ejemplo:

```
// varias acciones se llevan a cabo en las funciones "setup" y "loop"
// pero no se reporta ninguna información al programa principal.

void setup()
{
  // ...
}

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

3. FUNCIONES:

E/S DIGITALES

PINMODE()

Descripción

Configura el pin especificado para comportarse como una entrada o una salida. Mira la descripción de See the description of pines digitales para más información.

Sintaxis

pinMode(pin, modo)

Parametros

pin: el numero del pin que se desea configurar

modo: INPUT (Entrada) o OUTPUT (Salida)

Devuelve

Nada

Ejemplo

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

void setup()
{
  pinMode(ledPin, OUTPUT);      //configura el pin del LED como salida
}

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

Nota

Una entrada analógica puede ser usada como un pin digital, refiriéndose a ellos desde el número 14 (entrada analógica 0) a 19 (entrada analógica 5).

DIGITALWRITE()

Descripción

Escribe un valor HIGH o LOW hacia un pin digital.

Si el pin ha sido configurado como OUTPUT con pinMode(), su voltaje será establecido al correspondiente valor: 5V (o 3.3V en tarjetas de 3.3V) para HIGH, 0V (tierra) para LOW.

Si el pin es configurado como INPUT, escribir un valor de HIGH con digitalWrite() habilitará una resistencia interna de 20K conectada en pullup (ver el tutorial de pines digitales). Escribir LOW invalidará la resistencia. La resistencia es sufuciente para hacer brillar un LED de forma opaca, si los LEDs aparentan funcionar, pero no muy iluminados, esta puede ser la causa. La solución es establecer el pin como salida con la función pinMode().

NOTA: El pin digital número 13 es más difícil de usar que los otros pines digitales por que tiene un LED y una resistencia adjuntos, los cuales se encuentran soldados a la tarjeta, y la mayoría de las tarjetas se encuentran así. Si habilitas la resistencia interna en pullup, proporcionará 1.7V en vez de los 5V esperados, por que el LED soldado en la tarjeta y resistencias bajan el nivel de voltaje, significando que siempre regresará LOW. Si debes usar el pin número 13 como entrada digital, usa una resistencia externa conectada a pulldown.

Sintaxis

digitalWrite(pin, valor)

Parameters

pin: el número de pin

valor: HIGH o LOW

Devuelve

nada

Ejemplo

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

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

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

Establece el pin número 13 a HIGH, hace un retraso con la duración de un segundo, y regresa el pin a LOW.

Nota

Los pines analogicos pueden ser usados como pines digitales con los números 14 (entrada analogica número 0) hasta 19 (entrada analogica número 5).

DIGITALREAD()

Descripción

Lee el valor de un pin digital especificado, HIGH o LOW.

Sintaxis

digitalRead(pin)

Parámetros

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

Devuelve

HIGH o LOW

Ejemplo

```
int ledPin = 13; // LED conecado al pin digital número 13
int inPin = 7;   // botón (pushbutton) conectado al pin digital número 7
int val = 0;     // variable donde se almacena el valor leído

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

void loop()
{
  val = digitalRead(inPin);   // leer el pin de entrada
  digitalWrite(ledPin, val);    // establece el LED al valor del botón
}
```

Establece el pin número 13 al mismo valor que el pin número 7, que es una entrada.

Nota

Si el pin no esta conectado a algo, digitalRead() puede regresar HIGH o LOW (y esto puede cambiar aleatoriamente).

Los pines analogicos pueden ser usados como pines digitales con los números 14 (entrada analogica número 0) hasta 19 (entrada analogica número 5).

E/S Analógicas

ANALOGREAD()

Descripción

Lee el valor de tensión en el pin analógico especificado. La placa Arduino posee 6 canales (8 canalaes en el Mini y Nano y 16 en el Mega)conectados a un conversor analógico digital de 10 bits. Esto significa que convertirá tensiones entre 0 y 5 voltios a un número entero entre 0 y 1023. Esto proporciona una resolución en la lectura de: 5 voltios / 1024 unidades, es decir, 0.0049 voltios (4.9 mV)por unidad. El rango de entrada puede ser cambiado usando la función analogReference().

El conversor tarda aproximadamente 100 microsegundos (0.0001 segundos) en leer una entrada analógica por lo que se puede llevar una tasa de lectura máxima aproximada de 10.000 lecturas por segundo.

Sintaxis

analogRead(pin)

Parámetros

pin: Indica el número del pin de la entrada analógica que deseamos leer (0 a 5 en la mayoría de las placas, de 0 a 7 en las Mini y Nano y de 0 a 15 en las Mega)

Devuelve

int (0 a 1023)

Nota

Si la entrada analógica que vamos a leer no está conectada a nada, el valor que devolverá la función analogRead() fluctuará dependiendo de muchos valores (los valores de otras entradas analógicas, que tan cerca está tu mano de la entrada en cuestión, etc.)

Ejemplo

```
int analogPin = 3;        // el pin analógico 3 conectado al dial de un potenciómetro (terminal central del potenciómetro)
                      // los terminales exteriores del potenciómetro conectados a +5V y masa respectivamente
int val = 0;           // declaración de la variable en la que se almacenará el valor leído por el conversor.

void setup()
{
  Serial.begin(9600);          //  Inicialización del modulo Serial.
}

void loop()
{
  val = analogRead(analogPin);    // lee el valor de tensión del pin
  Serial.println(val);             // envía el valor leido vía serial.
}
```

ANALOGWRITE()

Descripción

Escribe un valor analógico (PWM) en un pin. Puede ser usado para controlar la luminosidad de un LED o la velocidad de un motor. Después de llamar a la función **analogWrite()**, el pin generará una onda cuadrada estable con el ciclo de trabajo especificado hasta que se vuelva a llamar a la función **analogWrite()** (o una llamada a las funciones **digitalRead()** o **digitalWrite()** en el mismo pin). La frecuencia de la señal PWM sera de aproximadamente 490 Hz.

En la mayoría de las placas Arduino (aquellas con el ATmega168 o ATmega328), se podrá generar señales PWM en los pines 3, 5, 6, 9, 10, y 11. En la placa Arduino Mega, se puede llevar a cabo con los pines desde el 2 hasta el pin 13. Las placas Arduino más antiguas que posean el chip ATmega8 solo podrán usar la función analogWrite() con los pines 9, 10 y 11. No hace faltar configurar el pin como salida para poder usar la función analogWrite().

La función *analogWrite* no tienen ninguna relación con los pines de entrada analógicos ni con la función *analogRead*.

Sintaxis

analogWrite(pin, valor)

Parámetros

pin: Es el pin en el cual se quiere generar la señal PWM.

valor: El ciclo de trabajo deseado comprendido entre 0 (siempre apagado) y 255 (siempre encendido).

Devuelve

Nada

Notas y problemas conocidos.

Las señales PWM generadas en los pines 5 y 6 poseerán ciclos de trabajo superiores a lo esperado. Esto es así por que

para esos dos pines se utiliza el mismo temporizador que se utiliza en las funciones millis() y delay(). Este efecto se notará mucho más en ciclos de trabajo bajos (por ejemplo de 0 a 10) y puede ser que aunque configuremos esos pines con una señal de ciclo de trabajo cero no llegue a ser verdaderamente 0.

Ejemplo

Produce una señal donde conectamos el LED, cuyo ciclo de trabajo es proporcional a la tensión leída en el potenciómetro.

```
int ledPin = 9;      // LED conectado al pin digital 9
int analogPin = 3;   // potenciómetro conectado al pin 3
int val = 0;         // variable en el que se almacena el dato leído

void setup()
{
  pinMode(ledPin, OUTPUT);   // sets the pin as output
}

void loop()
{
  val = analogRead(analogPin);   // lee la tensión en el pin
  analogWrite(ledPin, val / 4);  // los valores de analogRead van desde 0 a 1023
y los valores de analogWrite values van desde 0 a 255, por eso ajustamos el
ciclo de trabajo a el valor leído dividido por 4.
}
```

ANALOGWRITE()

Descripción

Escribe un valor analógico (PWM) en un pin. Puede ser usado para controlar la luminosidad de un LED o la velocidad de un motor. Después de llamar a la función **analogWrite()**, el pin generará una onda cuadrada estable con el ciclo de trabajo especificado hasta que se vuelva a llamar a la función **analogWrite()** (o una llamada a las funciones **digitalRead()** o **digitalWrite()** en el mismo pin). La frecuencia de la señal PWM sera de aproximadamente 490 Hz.

En la mayoría de las placas Arduino (aquellas con el ATmega168 o ATmega328), se podrá generar señales PWM en los pines 3, 5, 6, 9, 10, y 11. En la placa Arduino Mega, se puede llevar a cabo con los pines desde el 2 hasta el pin 13. Las placas Arduino más antiguas que posean el chip ATmega8 solo podrán usar la función analogWrite() con los pines 9, 10 y 11. No hace faltar configurar el pin como salida para poder usar la función analogWrite().

La función *analogWrite* no tienen ninguna relación con los pines de entrada analógicos ni con la función *analogRead*.

Sintaxis

analogWrite(pin, valor)

Parámetros

pin: Es el pin en el cual se quiere generar la señal PWM.

valor: El ciclo de trabajo deseado comprendido entre 0 (siempre apagado) y 255 (siempre encendido).

Devuelve

Nada

Notas y problemas conocidos.

Las señales PWM generadas en los pines 5 y 6 poseerán ciclos de trabajo superiores a lo esperado. Esto es así por que para esos dos pines se utiliza el mismo temporizador que se utiliza en las funciones millis() y delay(). Este efecto se notará mucho más en ciclos de trabajo bajos (por ejemplo de 0 a 10) y puede ser que aunque configuremos esos pines con una señal de ciclo de trabajo cero no llegue a ser verdaderamente 0.

Ejemplo

Produce una señal donde conectamos el LED, cuyo ciclo de trabajo es proporcional a la tensión leída en el potenciómetro.

```
int ledPin = 9;      // LED conectado al pin digital 9
int analogPin = 3;   // potenciómetro conectado al pin 3
int val = 0;         // variable en el que se almacena el dato leído

void setup()
{
  pinMode(ledPin, OUTPUT);   // sets the pin as output
}

void loop()
{
  val = analogRead(analogPin);   // lee la tensión en el pin
  analogWrite(ledPin, val / 4);  // los valores de analogRead van desde 0 a 1023 y los valores de analogWrite values van desde 0 a 255, por eso ajustamos el ciclo de trabajo a el valor leído dividido por 4.
}
```

E/S AVANZADAS

TONE()

Descripción

Genera una onda cuadrada de la frecuencia especificada (y un 50% de ciclo de trabajo) en un pin. La duración puede ser especificada, en caso contrario la onda continua hasta que haya una llamada a noTone(). El pin puede conectarse a un zumbador piezoeléctrico u otro altavoz que haga sonar los tonos.

Sólo puede generarse un tono cada vez. Si un tono está sonando en un pin diferente, la llamada a tone() no tendrá efecto. Si el tono está sonando en el mismo pin, la llamada establecerá la nueva frecuencia.

El uso de la función tone() interferirá con la salida PWM en los ins 3 y 11 (en otras placas distintas de la Mega).

NOTA: si quieres hacer sonar diferentes tonos en múltiples pins necesitas llamar a noTone() en un pin antes de llamar a tone() en el siguiente pin.

Sintáxis

tone(pin, frecuencia)
tone(pin, frecuencia, duracion)

Parámetros

pin: el pin en el que generar el tono

frecuencia: la frecuencia del tono en hercios.

duracion: la duración del tono en milisegundos (opcional)

NOTA: las frecuencias audibles por el oído humano van de los 20Hz a los 20KHz por lo que el parámetro "frecuencia" debería estar comprendido entre estos dos valores.

Devuelve

Nada

NOTONE()

Descripción

Detiene la generación de la señal cuadrada que se activa al hacer uso de la función tone(). No tiene efecto si no se está generando ningún tono.

NOTA: si quieres generar tonos diferentes en múltiples pines , necesitas usar la función noTone() en el pin antes de llamar a la función tone() en el siguiente pin.

Sintaxis

noTone(pin)

Parámetros

pin: el pin en el cual se va a parar de generar el tono.

Devuelve

Nada

SHIFTOUT()

Descripción

Desplaza un byte de datos bit a bit. Empieza desde el bit más significante (el de más a la izquierda) o el menos significante (el más a la derecha). Cada bit se escribe siguiendo su turno en un pin de datos, después de conmutar un pin de reloj (señal de reloj) para indicar que el bit está disponible.

Esto es conocido como protocolo serie síncrono y es la forma común que utilizan los microcontroladores para comunicarse con sensores y con otros microcontroladores. Ambos dispositivos permanecen sincronizados, y se comunican a velocidades cercanas a las máximas, hasta que ambos compartan la misma linea de reloj. En la documentación del harware interno de los chips se hace referencia a menudo a esta característica como Serial Peripheral Interface (SPI).

Sintaxis

shiftOut(pinDatos, pinReloj, ordenBits, valor)

Parametros

pinDatos: el pin en el cual extraer cada bit (*int*)

pinReloj: el pin que hay que conmutar cada vez que a un **pinDatos** le ha sido enviado el valor correcto (**int**)

ordenBits: en qué orden desplazar los bits; si hacia el **MSBFIRST** (bit más significante primero) o hacia el **LSBFIRST** (bit menos significante primero).
valor: los datos que rotar. (*byte*)

Retorno

Ninguno

Nota

El **pinDatos** y **pinReloj** deben estar ya configurados como salida con una llamada previa a pinMode().

shiftOut se encuentra actualmente configurada para extraer un byte (8 bits) por lo que necesita realizar una operación de dos pasos para extraer valores más grandes de 255.

```
// Haz esto para una comunicación serie MSBFIRST (primero el bit más significativo)
int datos = 500;
// rota el byte más alto
```

```
shiftOut(pinDatos, pinReloj, MSBFIRST, (datos >> 8));
// rota el byte más bajo
shiftOut(datos, pinReloj, MSBFIRST, datos);

// O haz esto para una comunicación serie LSBFIRST (primero el bit menos significativo)
datos = 500;
// rota el byte más bajo
shiftOut(pinDatos, pinReloj, LSBFIRST, datos);
// rota el bit más alto
shiftOut(pinDatos, pinReloj, LSBFIRST, (datos >> 8));
```

Ejemplo

Para ver el circuito asociado a este ejemplo, ver el tutorial para controlar un registro de desplazamiento 74HC595.

```
//*************************************************************//
//  Name    : shiftOutCode, Hello World                        //
//  Author  : Carlyn Maw,Tom Igoe                              //
//  Date    : 25 Oct, 2006                                     //
//  Version : 1.0                                              //
//  Notes   : Código para utilizar un registro de desplazamiento//
//          :  74HC595 para contar de 0 a 255                  //
//*************************************************************

//Pin conectado al pin ST_CP del 74HC595
int latchPin = 8;
//Pin connectado al pin SH_CP del 74HC595
int clockPin = 12;
////Pin conectado al pin DS del 74HC595
int dataPin = 11;

void setup() {
  // Configura como salida los pines que se direccionan en el bucle principal (loop)
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //rutina de conteo
  for (int j = 0; j < 256; j++) {
    //pone a nivel bajo el latchPin y lo mantienen a nivel bajo todo el tiempo que estés transmitindo
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    //vuelve a poner el latchPin a nivel alto para señalizar que
    //no sigue siendo necesario eschuchar más información
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

PULSEIN()

Descripción

Lee un pulso (ya sea HIGH —alto— o LOW —bajo—) en un pin. Por ejemplo, si **value** es **HIGH**, **pulseIn()** espera a que el pin sea **HIGH**, empieza a cronometrar, espera a que el pin sea **LOW** y entonces detiene la medida de tiempo.

Devuelve la anchura del pulso en microsegundos. Interrumpe la medida y devuelve 0 si el pulso no ha comenzado en un tiempo especificado.

La medida del tiempo en esta función ha sido determinada de forma empírica y está sujeta a errores en pulsos largos. Funciona correctamente en pulsos con una anchura de 10 microsegundos a tres minutos.

Sintaxis

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

Parámetros

pin: el número del pin en el que se realiza la medida del pulso. (*int*)

value: tipo de pulso. Puede ser HIGH o LOW. (*int*)

timeout (opcional): el tiempo en microsegundos máximo a esperar antes de que se inicie el pulso. (*unsigned long*)

Devuelve

el ancho del pulso (en microsegundos) ó 0 si el pulso no ha empezado antes del timeout (*unsigned long*)

Ejemplo

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

void setup()
{
  pinMode(pin, INPUT);
}

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

TIEMPO

MIN(X, Y)

Descripción

Calcula el mínimo de dos números.

Parametros

x: el primer número, cualquier tipo de dato

y: el segundo número, cualquier tipo de dato

Devuelve

El más pequeño entre los dos números.

Ejemplos

```
sensVal = min(sensVal, 100); // asigna sensVal al menor entre sensVal y 100
                             // asegurando que nunca pasará de 100.
```

Nota

La función max() es usualmente usada para limitar el límite inferior del rango de una variable, mientras que la función min() es usada para limitar el límite superior del rango de una variable.

Advertencia

Debido a la forma en la que se ha implementado la función min(), evite realizar operaciones en el argumentos de la función ya que podría producir errores.

```
min(a++, 100);   // evite esto - puede producir errores

a++;
min(a, 100);     // use este método en cambio - mantenga cualquier operación fuera de los paréntesis
```

MAX(X, Y)

Descripción

Calcula el máximo de dos números.

Parámetros

x: El primer número, cualquier tipo de dato.

y: El segundo número, cualquier tipo de dato.

Devuelve

El mayor de ambos parámetros.

Ejemplo

```
sensVal = max(senVal, 20); // asigna a sensVal su propio valor o, de ser superior, 20.
                           // (una forma efectiva de asegurarse que el valor mínimo de senVal sea 20)
```

Nota

max() suele ser usado para restringir el valor más bajo del rango de una variable, mientras que *min()* es utilizado para restringir el valor máximo del rango.

Atención

Debido a la forma en que la función *max()* es implementada, debes evitar usar otras funciones al definir los parámetros, puede derivar en resultados incorrectos.

```
max(a--, 0);   // evitar esto - puede dar resultados incorrectos.

a--;           // en su lugar, hacerlo así -
max(a, 0);     // mantener cualquier operación fuera de los paréntesis.
```

ABS(X)

Descripción

Calcula el valor absoluto de un número.

Parámetros

x: El numero cuyo valor absoluto deseamos calcular

Devuelve

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

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

Precaución

Debido a la forma en la que se ha implementado la función abs(), evite usar otras funciones como parámetro debido a que puede ocasionar que se devuelva un resultado incorrecto.

```
abs(a++);   // evite esto - produce errores en el resultado

a++;          // hazlo de esta manera -
abs(a);       // mantenga cualquier otra operación fuera de la función
```

CONSTRAIN(X, A, B)

Descripción

Restringe un número a un rango definido.

Parámetros

x: el número a restringir, cualquier tipo de datos.

a: El número menor del rango, cualquier tipo de datos.

b: El número mayor del rango, cualquier tipo de datos.

Retorna

x: si **x** está entre **a** y **b**

a: si **x** es menor que **a**

b: si **x** es mayor que **b**

Ejemplo

```
sensVal = constrain(sensVal, 10, 150);
// limita el valor del sensor entre 10 y 150
```

MAP(VALUE, FROMLOW, FROMHIGH, TOLOW, TOHIGH)

Descripción

Re-mapea un número desde un rango hacia otro. Ésto significa que, un valor (*value*) con respecto al rango *fromLow-fromHight* será mapeado al rango *toLow-toHigh*.

No se limitan los valores dentro del rango, ya que los valores *fuera de rango* son a veces objetivos y útiles. La función **constrain()** puede ser usada tanto antes como después de ésta función, si los límites de los rangos son deseados.

Ten en cuenta que los límites "inferiores" de algún rango pueden ser mayores o menores que el límite "superior" por lo

que *map()* puede utilizarse para revertir una serie de números, por ejemplo:

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

La función maneja correctamente también los números negativos, por ejemplo:

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

también es válido y funciona correctamente.

La función *map()* usa matemática de enteros por lo que no generará fracciones, aunque fuere el resultado correcto. Los resultados en fracciones se truncan, y no son redondeados o promediados.

Parámetros

value: el número (valor) a mapear.

fromLow: el límite inferior del rango actual del valor.

fromHigh: el límite superior del rango actual del valor.

toLow: límite inferior del rango deseado.

toHigh: límite superior del rango deseado.

Devuelve

El valor mapeado.

Ejemplo

```
/* Mapea un valor análogo a 8 bits (0 a 255) */
void setup() {}

void loop()
{
  int val = analogRead(0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

Apéndice

Para los interesados en el funcionamiento de la función, aquí está su código:

```
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;
}
```

POW(BASE, EXPONENTE)

Descripción

Calcula el valor de un número elevado a otro número. Pow() puede ser usado para elevar un número a una fracción. Esta función es útil para generar datos exponenciales o curvas.

Parámetros

base: base que queremos elevar (*float*) (Coma flotante)

exponente: la potencia a la que se quiere elevar la base (*float*) (Coma flotante)

Devuelve

El resultado de la exponenciación (*double*)(Doble)

Ejemplo

Mira la función fscale en la librería del código.

SQ(X)

Descripción

Calcula el cuadrado de un numero: el numero multiplicado por el mismo.

Parámetros

x: el numero, cualquier tipo de dato

Devuelve

El cuadrado del numero

SQRT(X)

Descripción

Calcula la raíz cuadrada de un numero.

Parámetros

x: el numero, cualquier tipo de dato

Devuelve

doble, la raíz cuadrada del numero.

Trigonometría

SIN(RAD)

Descripción

Calcula el seno de un ángulo (en raianes). El resultado será entre -1 y 1.

Parametros

rad: el ángulo en radianes (*float*)

Retorno

El seno del ángulo (*double*)

Nota

Serial.print() y Serial.println() no soportan la impresión de valores de tipo *float*.

COS(RAD)

Descripción

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

Parámetros

rad: el ángulo en radianes (*float*)

Retorna

El coseno del ángulo ("double")

Nota

Serial.print() y Serial.println() no soportan actualmente la impresión de números con decimales.

TAN(RAD)

Descripción

Calcula la tangente de un ángulo (en radianes). El resultado estará entre el menos infinito y el infinito.

Parámetros

rad: el ángulo en radianes (*float*)

Retorno

La tangente del ángulo (*double*)

Nota

Serial.print() y Serial.println() no soportan actualmente imprimir variales de tipo float.

Números Aleatorios

RANDOMSEED(SEED)

Descripción

randomSeed() inicializa el generador de números pseudoaleatorios, haciéndole empezar en un punto arbitrario de su secuencia aleatoria. Esta secuencia, aunque muy larga y aleatoria, es siempre la misma.

Si es importante que la secuencia de valores generada por random() difiera en ejecuciones sucesivas de un programa, es recomendable utilizar randomSeed() (*seed* en inglés, semilla) para inicializar el generador de números aleatorios con una entrada mínimamente aleatoria como analogRead() en un pin desconectado.

No obstante, puede ser útil usar secuencias pseudoaleatorias que se repitan exactamente. Esto se consigue llamando a randomSeed() con un número fijo antes de empezar la generación de la secuencia.

Parámetros

long, int - recibe un número para generar la semilla.

Devuelve

no devuelve nada

Example

```
long numAleatorio;

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

void loop(){
  randNumber = random(300);
```

```
  Serial.println(numAleatorio);

  delay(50);
}
```

RANDOM()

Descripción

La función *random* genera números pseudoaleatorios.

Sintaxis

random(max)
random(min, max)

Parámetros

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

max - límite superior del valor aleatorio, exclusive (se devuelve hasta el anterior)

Devuelve

un número aleatorio entre min y max (*long*)

Nota:

Si es importante que la secuencia de valores generada por random() difiera en ejecuciones sucesivas de un programa, es recomendable utilizar randomSeed() (*seed* en inglés, semilla) para inicializar el generador de números aleatorios con una entrada mínimamente aleatoria como analogRead() en un pin desconectado.

No obstante, puede ser útil usar secuencias pseudoaleatorias que se repitan exactamente. Esto se consigue llamando a randomSeed() con un número fijo antes de empezar la generación de la secuencia.

Ejemplo

```
long numAleatorio;

void setup(){
  Serial.begin(9600);
  // si la entrada analógica 0 no está conectada,
  // la llamada a randomSeed() recibirá ruido estático
  // (analógico) y se generarán diferentes semillas
  // cada vez que se ejecute el sketch.
  randomSeed(analogRead(0));
}

void loop() {
  // escribe un número aleatorio de 0 a 299
  numAleatorio = random(300);
  Serial.println(numAleatorio);

  // escribe un número aleatorio de 10 a 19
  numAleatorio = random(10, 20);
  Serial.println(numAleatorio);

  delay(50);
}
```

COMUNICACIÓN

SERIAL

Se utiliza para la comunicación entre la placa Arduino y un ordenador u otros dispositivos. Todas las placas Arduino tienen al menos un puerto serie (también conocido como UART o USART): **Serial**. Se comunica a través de los pines digitales 0 (RX) y 1 (TX), así como con el ordenador mediante USB. Por lo tanto, si utilizas estas funciones, no puedes usar los pines 0 y 1 como entrada o salida digital.

Puedes utilizar el monitor del puerto serie incorporado en el entorno Arduino para comunicarte con la placa Arduino. Haz clic en el botón del monitor de puerto serie en la barra de herramientas y selecciona la misma velocidad en baudios utilizada en la llamada a begin().

La placa Arduino Mega tiene tres puertos adicionales de serie: **Serial1** en los pines 19 (RX) y 18 (TX), **Serial2** en los pines 17 (RX) y 16 (TX), **Serial3** en los pines 15 (RX) y 14 (TX). Para utilizar estos pines para comunicarse con el ordenador personal, necesitarás un adaptador USB adicional a serie, ya que no están conectados al adaptador USB-Serie de la placa Arduino Mega. Para usarlos para comunicarse con un dispositivo serie externo TTL, conecta el pin TX al pin RX del dispositivo, el RX al pin TX del dispositivo, y el GND de tu Arduino Mega a masa del dispositivo. (No conectes estos pines directamente a un puerto serie RS232, que operan a +/- 12V y esto puede dañar la placa Arduino.)

Funciones

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

Funciones

BEGIN()

Descripción

Establece la velocidad de datos en bits por segundo (baudios) para la transmisión de datos en serie. Para comunicarse con el computador, utilice una de estas velocidades: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 o 115200. Sin embargo, puedes especificar otras velocidades - por ejemplo, para comunicarte a través de los pines 0 y 1 con un componente que requiere una velocidad de transmisión en particular.

Sintaxis

Serial.begin(speed)

Solamente en Arduino Mega:
Serial1.begin(speed)
Serial2.begin(speed)
Serial3.begin(speed)

Parámetros

speed: Velocidad en bits por segundo (baudios) - *long*

Devuelve

nada

Ejemplo:

```
void setup() {
        Serial.begin(9600);     // abre el puerto serie y establece la velocidad
en 9600 bps
}

void loop() {}
```

Ejemplo para Arduino Mega:

```
// Arduino Mega usando sus 4 puertos serie
// (Serial, Serial1, Serial2, Serial3),
// con diferentes velocidades de datos:

void setup(){
  Serial.begin(9600);
  Serial1.begin(38400);
  Serial2.begin(19200);
  Serial3.begin(4800);

  Serial.println("Hola ordenador");
  Serial1.println("Hola Serial 1");
  Serial2.println("Hola Serial 2");
  Serial3.println("Hola Serial 3");
}

void loop() {}
```

END()

Descripción

Desactiva la comunicación serie, permitiendo a los pines RX and TX ser usados como entradas o salidas digitales. Para reactivar la comunicación serie, llama al método Serial.begin().

Sintaxis

Serial.end()

Solamente en Arduino Mega:

Serial1.end()
Serial2.end()
Serial3.end()

Parámetros

ninguno

Devuelve

nada

AVAILABLE()

Descripción

Devuelve el número de bytes (caracteres) disponibles para ser leídos por el puerto serie. Se refiere a datos ya recibidos y disponibles en el buffer de recepción del puerto (que tiene una capacidad de 128 bytes).

Sintaxis

Serial.available()

Sólo para Arduino Mega:

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

Parametros

ninguno

Devuelve

el número de bytes disponibles para ser leídos

Ejemplo

```
int incomingByte = 0;   // para los datos de entrada serie

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

void loop() {

        // envía datos solo cuando recibe datos:
        if (Serial.available() > 0) {
                // lee el byte de entrada:
                incomingByte = Serial.read();

                // muestra lo que tiene:
                Serial.print("He recibido: ");
                Serial.println(incomingByte, DEC);
        }
}
```

Ejemplo Arduino Mega:

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

}

void loop() {
  // lee desde el puerto 0, envía al puerto 1:
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, BYTE);

  }
  // lee del puerto 1, envía al puerto 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, BYTE);
  }
}
```

READ()

Descripción

Lee los datos entrantes del puerto serie.

Sintaxis

Serial.read()

Solamente en Arduino Mega:

Serial1.read()
Serial2.read()
Serial3.read()

Parámetros

Ninguno

Devuelve

el primer byte disponible recibido por el puerto serie (devuelve -1 si no hay datos disponibles) - *int*

Ejemplo

```
int incomingByte = 0;   // para el byte leido

void setup() {
        Serial.begin(9600);     // abre el puerto serie a 9600 bps
}

void loop() {

        // envia datos solamente cuando recibe datos
        if (Serial.available() > 0) {
                // lee el byte entrante:
                incomingByte = Serial.read();

                // dice lo que ha recibido:
                Serial.print("He recibido: ");
                Serial.println(incomingByte, DEC);
        }
}
```

FLUSH()

Descripción

Vacía el búfer de entrada de datos en serie. Es decir, cualquier llamada a Serial.read () o Serial.available () devolverá sólo los datos recibidos después la llamada más reciente a Serial.flush ().

Sintaxis

Serial.flush()

Solamente en Arduino Mega:

Serial1.flush()
Serial2.flush()
Serial3.flush()

Parámetros

ninguno

Retorna

nada

PRINT()

Descripción

Imprime los datos al puerto serie como texto ASCII. Este comando puede tomar muchas formas. Los números son impresos mediante un juego de caracteres ASCII para cada dígito. Los valores de tipo "float" son impresos en forma de dígitos ASCII con dos decimales por defecto. Los valores tipo "byte" se envían como un único carácter. Los caracteres y las cadenas se envían tal cual. Por ejemplo:

- Serial.print(78) imprime "78"
- Serial.print(1.23456) imprime "1.23"
- Serial.print(byte(78)) imprime "N" (cuyo código ASCII es 78)
- Serial.print('N') imprime "N"
- Serial.print("Hello world.") imprime "Hello world."

Un segundo parámetro opcional especifica la base (formato) a usar; los valores permitidos son BYTE, BIN (binarios o base 2), OCT (octales o base 8), DEC (decimales o base 10), HEX (hexadecimales o base 16). Para números de coma flotante, este parámetro especifica el numero de posiciones decimales a usar. Por ejemplo:

- Serial.print(78, BYTE) imprime "N"
- Serial.print(78, BIN) imprime "1001110"
- Serial.print(78, OCT) imprime "116"
- Serial.print(78, DEC) imprime "78"
- Serial.print(78, HEX) imprime "4E"
- Serial.println(1.23456, 0) imprime "1"
- Serial.println(1.23456, 2) imprime "1.23"
- Serial.println(1.23456, 4) imprime "1.2346"

Sintaxis

Serial.print(val)
Serial.print(val, format)

Parámetros

val: el valor a imprimir - de cualquier tipo

format: especifica el número de la base (para números enteros) o el número de posiciones decimales (para números de coma flotante o tipo "float")

Devuelve

Nada

Ejemplo:

```
/*Usa un bucle FOR para los datos e imprime un número en varios formatos.*/
int x = 0;    // variable

void setup() {
  Serial.begin(9600);      // abre el puerto serie a 9600 bps:
}
```

```
void loop() {
  // print labels
  Serial.print("SIN FORMATO");       // imprime un texto
  Serial.print("\t");              // imprime un tabulado

  Serial.print("DEC");
  Serial.print("\t");

  Serial.print("HEX");
  Serial.print("\t");

  Serial.print("OCT");
  Serial.print("\t");

  Serial.print("BIN");
  Serial.print("\t");

  Serial.println("BYTE");

  for(x=0; x< 64; x++){    // solo una parte de la tabla

    // imprime en varios formatos:
    Serial.print(x);       // imprime como codificado ASCII decimal - igual que
"DEC"
    Serial.print("\t");    // imprime un tabulado

    Serial.print(x, DEC);  // imprime como codificado ASCII decimal
    Serial.print("\t");    // imprime un tabulado

    Serial.print(x, HEX);  // imprime como codificado ASCII hexadecimal
    Serial.print("\t");    // imprime un tabulado

    Serial.print(x, OCT);  // imprime como codificado ASCII octal
    Serial.print("\t");    // imprime un tabulado

    Serial.print(x, BIN);  // imprime como codificado ASCII binario
    Serial.print("\t");    // imprime un tabulado

    Serial.println(x, BYTE);    // imprime el valor en bruto del byte,
    //                             y añade el salto de línea con "println"
    delay(200);            // espera 200 milisegundos
  }
  Serial.println("");      //imprime otro salto de línea
}
```

Sugerencias de programación / Problemas conocidos

El último carácter a imprimir se transmite a través del puerto serie después de que Serial.print () ha regresado.

PRINTLN()

Descripción

Imprime los datos al puerto serie como texto ASCII seguido de un retorno de carro (ASCII 13, o '\r') y un carácter de avance de línea (ASCII 10, o '\n'). Este comando tiene la misma forma que Serial.print ().

Sintaxis

Serial.println(val)
Serial.println(val, format)

Parámetros

val: el valor a imprimir - de cualquier tipo

format: especifica el número de la base (para números enteros) o el número de posiciones decimales (para números de coma flotante o tipo "float")

Devuelve

Nada

Ejemplo:

```
/*
  Analog input

 Lee el pin analógico 0, e imprime su valor por el puerto serie.

 created 24 March 2006
 by Tom Igoe
 */

int analogValue = 0;    // variable para guardar el valor analogico

void setup() {
  // abre el puerto serie a 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // lee la entrada analogica en el pin 0:
  analogValue = analogRead(0);

  // imprime el valor en varios formatos:
  Serial.println(analogValue);       // imprime como ASCII decimal
  Serial.println(analogValue, DEC);  // imprime como ASCII decimal
  Serial.println(analogValue, HEX);  // imprime como ASCII hexadecimal
  Serial.println(analogValue, OCT);  // imprime como ASCII octal
  Serial.println(analogValue, BIN);  // imprime como ASCII binario
  Serial.println(analogValue, BYTE); // imprime el valor del byte

  // espera 10 milisegundos antes de la siguiente lectura.
  delay(10);
}
```

WRITE()

Descripción

Escribe datos binarios en el puerto serie. Estos datos se envían como un byte o una serie de bytes; para enviar los caracteres que representan los dígitos de un número usar función print () en su lugar.

Syntax

Serial.write(val)
Serial.write(str)
Serial.write(buf, len)

Arduino Mega tambien soporta: Serial1, Serial2, Serial3 (en lugar de Serial)

Parámetros

val: un valor para enviar como un solo byte

str: una cadena 'string' para enviar como una serie de bytes

buf: un 'array' para enviar como una serie de bytes

len: longitud del búfer