MANUAL DE ARDUINO

PROGRAMACIÓN Y CONCEPTOS BÁSICOS

barGraph

```
const int analogPin = A0;    // the pin that the potentiometer is attached to
const int ledCount = 10;     // the number of LEDs in the bar graph

int ledPins[] = {
  2, 3, 4, 5, 6, 7,8,9,10,11 };   // an array of pin numbers to which LEDs are attached


void setup() {
  // loop over the pin array and set them all to output:
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT);
  }
}

void loop() {
  // read the potentiometer:
  int sensorReading = analogRead(analogPin);
  // map the result to a range from 0 to the number of LEDs
  int ledLevel = map(sensorReading, 0, 1023, 0, ledCount);

  // loop over the LED array:
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    // if the array element's index is less than ledLevel,
    // turn the pin for this element on:
    if (thisLed < ledLevel) {
      digitalWrite(ledPins[thisLed], HIGH);
```

Maker Zona

Este manual aborda todos los conceptos básicos de Arduino y es una excelente guía para todo aquel que quiera iniciarse en este apasionante mundo.

El manual ha sido confeccionado por Raúl Diosdado usando para ello la siguiente información y recursos:

http://www.arduino.cc (Pagina oficial de Arduino)

Fritzing (Elaboración de esquemáticos y montajes) http://fritzing.org

http://www.zonamaker.com (Recursos propios)

Un profesor solía decirme, "*No inventes la rueda, eso ya se preocupo alguien en inventarla, tu tan solo úsala*" Haced que Arduino sea vuestra rueda

Primera edición publicada en Septiembre de 2014

INDICE

1. Estructura de un programa

La estructura de un programa en Arduino puede llegar a ser muy diferente en función de la complejidad de la aplicación que queramos crear, pero como en la mayoría de lenguajes de programación esta estructura está formada por funciones, sentencias, bucles y otros elementos que conforman la estructura de programa.

1.1 Funciones

Una función es un bloque de código con un nombre y un conjunto de estamentos que son ejecutados cuando se llama a dicha función. Las funciones de usuario se crean para realizar tareas repetitivas reduciendo de esa manera el tamaño del programa.

Las funciones están asociadas a un tipo de valor "type", este es el valor que devolverá la función una vez termine su ejecución.

```
type nombreFunción (parámetros)
{
  Estamentos o instrucciones;
}
```

Si la función no devuelve ningún valor, se le asignará el tipo "void" o función vacía. No es obligatorio pasarles parámetros a la función, se pueden crear funciones independientes que obtengan sus propios parámetros para trabajar.

Ejemplo de función:

```
/* Define la función "multiplicar" a la que se le pasarán los 2 números que
se deseen multiplicar devolviendo un valor entero (int). */

int multiplicar (int x, int y) //función multiplicar
{
  int resultado;  //declara la variable donde se almacena el resultado
  resultado = x * y;  //ejecuta la operación de multiplicar
  return resultado;  //devuelve el resultado de la multiplicación
}
```

Para hacer uso de la función anterior, tan solo habrá que llamarla desde cualquier parte del programa. Esta llamada a la función se realiza de la siguiente manera:

```
void loop()
{
  int i =2;
```

```
  int j =3;
  int k;

   k = multiplicación(i, j);  //llama a la función multiplicación pasándole los
                              //parámetros "i" y "j"
}
```

En Arduino los programas pueden estar compuestos por infinidad de funciones, pero existe una estructura básica que deben de cumplir todos los programas. Esta estructura básica, está formada por dos funciones totalmente necesarias para la ejecución del programa, la función *setup()* y *loop()*.

1.1.1 Setup ()

La función *setup()* se invoca una sola vez al comienzo del programa. Esta función se usa para realizar la configuración inicial, dentro de esta configuración podemos establecer el modo de trabajo de los pines o inicializar la comunicación serie entre otras cosas.

```
void setup()
{
  Estamentos o instrucciones;
}
```

1.1.2 Loop ()

La función *loop()* es la función principal dentro del programa. Esta función se va a ejecutar continuamente de manera cíclica, ejecutando todas las instrucciones que se encuentren en su interior.

```
void loop()
{
  Estamentos o instrucciones;
}
```

Además de las funciones principales, existen otros elementos de la estructura de programa que ayudan a definir, delimitar, estructurar y a hacer más claro el contenido del programa.

1.2 Llaves {}

Las llaves definen el principio y el final de un bloque de instrucciones. Se usan para delimitar el inicio y fin de funciones como *setup()*o para delimitar el alcance de los bucles y condicionales del programa.

```
funcion()
{
  Estamentos o instrucciones;
}
```

1.3 Punto y coma ;

El punto y coma ";" se utiliza para definir el final de una instrucción y separarla de la siguiente. Si no colocamos punto y coma, el programa va a interpretar mal las instrucciones y se va a producir un error de compilación.

```
digitalWrite (10, HIGH);
```

El error más común a la hora de programar suele ser olvidar poner punto y coma al final de la instrucción.

1.4 Bloque de comentarios /*...*/

Los bloques de comentarios son áreas de texto que nos ayudan a describir o comentar un programa, estos bloques serán ignorados a la hora de compilar el programa en nuestro Arduino.

```
/* El bloque de comentario ayuda
al programador a describir el programa
*/
```

Se pueden introducir todas las líneas de texto que se deseen siempre que se encuentren entre los caracteres **/*...*/**.

Se recomienda el uso de bloques de comentarios siempre que se pueda, ya que ayudan a la comprensión del programa a personas ajenas al mismo, además, estos comentarios no van a ocupar espacio de programa, ya que son ignorados a la hora de compilar.

1.5 Línea de comentarios //

La línea de comentarios tienen la misma función que los bloques de comentarios, la única diferencia es que las líneas de comentarios suelen usarse para comentar instrucciones ya que solo afectan a una línea.

```
int x = 10;   //declara la variable 'x' como tipo entero de valor 13
```

2. Variables

Las variables son elementos donde se almacenan valores numéricos que serán usados por el programa. Como su nombre indica, las variables van a cambiar de valor con la evolución del programa y nos van a permitir crear la lógica del programa en función de estos cambios.

```
int variable_entrada = 0;   //declara una variable y le asigna el valor 0
variable_entrada = analogRead(2);    //la variable toma el valor de la entrada analógica 2
```

Una variable debe ser declarada y opcionalmente asignarle un valor antes de ser usada por el programa, si no hemos declarado la variable o lo hemos hecho con posterioridad a su uso va a dar un error al compilar el programa.

Es recomendable asignarle un valor inicial a las variables para no tener un valor indeterminado que pudiera ocasionar algún error a la hora de la ejecución del programa.

Cuando se asignen nombres a las variables, se deben de usar nombres que identifiquen claramente a dichas variables, para ello se usan nombres como "pulsador", "led", "entrada_1".... esto va a ayudar a hacer un código legible y fácil de entender.

2.1 Utilización de las variables

Las variables pueden ser declaradas en diferentes lugares del programa, en función del lugar donde sean declaradas, las variables van a ser de tipo global o local, determinando esto el ámbito de aplicación o la capacidad de ciertas partes del programa para hacer uso de las mismas.

Una variable global es aquella que puede ser vista y utilizada por cualquier función yestamento de un programa. Las variables globales se declaran al comienzo del programa, antes de la función *setup()*.

Una variable local es aquella que se define dentro de una función o como parte de un bucle. Solo será visible y podrá utilizarse dentro de la función o bucle donde es declarada.

La existencia de variables globales y locales, permite el uso de variables con el mismo nombre en partes diferentes del programa, estas variables podrán almacenar valores distintos sin que se produzca ningún conflicto, ya que a las variables locales solo tendrán acceso las funciones donde se declaren dichas variables.

Ejemplo de variables globales y locales:

```
int led = 10;   //"led" es una variable global visible para cualquier función
void setup()
{
  pinMode (led, OUTPUT);   //establece el pin "led" como salida
}
void loop ()
{
  float valor;   //"valor" es una variable local solo visible dentro de "loop"
  for (int i=0; i<20; i++)   //"i" es una variable local usada por el bucle for
  {
    digitalWrite (led, HIGH);  //uso de la variable global "led"
  }
}
```

2.2 Tipos de variables

A la hora de usar una variable dentro del programa debemos de pararnos a pensar que información o qué tipo de operaciones vamos a aplicar a estas variables y en función de esto asignarle un tipo u otro. Debemos de asegurarnos que el tipo de variable que estemos usando tiene rango suficiente para almacenar los datos sin llegar a ser desproporcionada, ya que usar variables grandes va a consumir más recursos de memoria.

Tipo de variables	Memoria que ocupa	Rango de valores
boolean	8 bit's	0 o 1 (true o false)
char*	8 bit's	-128 a 127
int	16 bit's	-32.768 a 32.767
unsigned int	16 bit's	0 a 65.535
long	32 bit's	-2.146.483.648 a 2.147483.647
unsigned long	32 bit's	0 a 4.294.967.295
float*	32 bit's	-3'4028235 E+38 a 3'4028235 E+38

* char: el tipo "char" se utiliza para almacenar caracteres en formato ASCII
* float: el tipo "float" se utiliza para datos con decimales, aunque cuenta con 32bit's, tan solo tiene una precisión de 7 dígitos en sus decimales.

3. Constantes

La diferencia fundamental entre una variable y una constante es que la constante va a ser un valor de solo lectura que no va a poder ser modificado con la evolución del programa.

Si queremos definir una constante, podemos hacerlo de manera muy similar a como definíamos las variables, tan solo hay que indicar al programa que se trata de un valor "no modificable", para ello hay que añadir antes del "tipo" la palabra "*const*", que va a hacer que la variable sea de solo lectura.

```
const float pi = 3.1415;    //crea una constante de tipo "float" y le asigna el valor 3.1415
```

Es posible crear constantes usando *#define*, esta sentencia va a nombrar las constantes antes de ser compiladas por Arduino. Las constantes creadas de esta manera no van a ocupar memoria de programa, ya que el compilador va a remplazar estas constantes con el valor definido en el momento de compilar.

```
#define pi 3.1415  //crea una constante usando #define a la que le asigna el valor 3.1415
```

Hay que tener cuidado a la hora de usar *#define*, ya que la sintaxis es diferente, no hay que poner el operador de asignación "=" ni cerrar sentencia con el punto y coma ";".

Usar #define puede ser problemático, ya que si existe en cualquier parte del programa alguna constante o variable con el mismo nombre que tengamos asignado en #define, el programa lo va a sustituir con el valor que tengamos asignado a esta constante, por ello se debe tener cuidado a la hora de asignar constantes con *#define* y no crear constantes muy genéricas como *#define x 10,*ya que "x" es muy común usarlo como variable dentro de alguna función o bucle.

4. Operadores

Los operadores son los elementos con los que vamos transformar las variables del programa, hacer comparaciones, contar un número determinado de eventos... .Los operadores se pueden considerar como uno de los elementos más importantes junto con las estructuras de control. Dentro de los operadores, podemos encontrarlos de varios tipos.

4.1 Operadores aritméticos

Los operadores aritméticos son operadores que nos van a permitir realizar operaciones básicas como sumar, restar, dividir, multiplicar...

```
x = x + 5;   //suma x+5 y guarda el valor en x
y = y - 8;   //resta 8 a el valor "y" almacena el resultado en "y"
z = z *  2;  //multiplica z*2 y guarda el valor en z
k = k /  3;  //divide k entre 3 y guarda el valor en k
p = 10;      // asigna a p el valor 10
```

Hay que tener en cuenta el tipo de variable que estamos usando a la hora de realizar operaciones, ya que si vamos a efectuar una operación que da como resultado un numero con decimales y hemos definido estas variables como *int* (entero), el resultado no va a mostrar la parte decimal (10 / 3 va a devolver 3 en lugar de 3.33).

4.2 Asignaciones compuestas

Las asignaciones compuestas combinan una operación aritmética con una variable asignada. Estas asignaciones son usadas comúnmente en bucles.

```
x ++;     //equivale a x = x + 1 (incrementa x en 1)
x --;     //equivale a x = x - 1 (decrementa x en 1)
x += y;   //equivale a x = x + y
x -= y;   //equivale a x = x - y
x * = y;  //equivale a x = x * y
x / = y;  //equivale a x = x / y
```

4.3 Operadores de comparación

Los operadores de comparación se usan con frecuencia en estructuras condicionales para comprobar si una condición se cumple o no. Normalmente estos condicionales realizan la comparación entre las variables y constantes del programa.

```
x == y;   //x es igual a y
x != y;   //x es distinto de y
x < y;    //x es menor que y
x > y;    //x es mayor que y
x <= y;   //x es menor o igual que y
x >=y;    // x es mayor o igual que y
```

4.4 Operadores Boleanos

Son operadores lógicos que se usan para comparar 2 o más expresiones y que no devuelven un valor, sino que dan un estado de “verdadero” (si se cumple la expresión) o “falso” (si no se cumple). Existen 3 operadores lógicos, AND "&&", OR "||" y NOT "!".

```
if (x<3 && x>0)    //Cierto si se cumplen las dos expresiones
if (x<7 || x=20)   //Cierto si se cumple alguna de las dos expresiones
if (!x=3)          //Cierto si x es distinto de 3
```

5. Estructuras de control

Dentro de las estructuras de control se engloban todos los estamentos que sirven para guiar al programa en una u en otra dirección en función de si se cumplen o no las condiciones que le marquemos al programa. Dentro de estas estructuras podemos encontrar condicionales, bucles o elementos de control de flujo.

5.1 Condicionales

Los condicionales son elementos que chequean un estado o condición y si esta condición se cumple se pasa a ejecutar las sentencias englobadas dentro de la condición.

5.1.1 If (si...)

If es un estamento que se utiliza para comprobar si una determinada condición se cumple. Si la condición se cumple, se pasará a ejecutar las sentencias encerradas dentro del *bloque if*, si no se cumple la condición, el programa saltará este bloque sin ejecutar ninguna instrucción.

```
if (x==10)  //Si x es igual a 10 ejecuta la instrucción
{
  ejecuta instrucciones;
}
```

5.1.2 If... else (si... si no...)

*If... else*funciona de igual forma que *if*, pero añade la posibilidad de que la condición no se cumpla,pasando a ejecutar las instrucciones encerradas dentro de *else*.

```
if (y != 10)  // Si "y" es distinto de 10 ejecuta las instrucciones
{
  ejecuta instrucciones;
}
else  //si no, ejecuta esta instrucción
{
  ejecuta instrucciones;
}
```

Este condicional puede ir precedido de otras estructuras condicionales del mismo tipo, anidando unas dentro de otras y haciendo que sean mutuamente excluyentes.

```
if (valor < 100)  // Si valor es menor que 100 ejecuta la instrucción
{
   ejecuta instrucciones;
}
else if (valor >=500)  // Si valor es mayor o igual que 500 ejecuta la instrucción
{
  ejecuta instrucciones;
}
else  //si no se cumplen las condiciones anteriores, ejecuta esta instrucción
{
  ejecuta instrucciones;
}
```

5.2 Bucles

Los bucles son elementos que hacen que el programa entre en un ciclo de repetición mientras se cumplan las condiciones del bucle.

5.2.1 For

El bucle *for* se usa para repetir un bloque de sentencias un número determinado de veces. Cada vez que se terminan de ejecutar las sentencias encerradas dentro del bucle, se comprobará la condición inicial, repitiéndose el buche mientras se cumpla dicha condición.

```
for (inicialización; condición; expresión)
{
  ejecuta instrucciones;
}
```

El buche *for* está formado por 3 partes, la inicialización, la condición del bucle y una expresión (la expresión no es necesaria o puede ponerse dentro de las sentencias del bucle).

Ejemplo de bucle *for*:

```
for (int x=0; x<10; x++)  //declara la variable x y la inicializa a 0, comprueba la
                          //condición (x<10), incrementa x en 1
{
  digitalWrite (13, HIGH);  //envía un 1 al pin 13
  delay (500);              //espera 500ms
  digitalWrite (13, LOW);  //envía un 0 al pin 13
  delay (500);              //espera 500ms
}
```

5.2.2 While

El bucle *while* es un bucle que se repetirá constantemente mientras se cumpla la expresión del bucle. Dentro de la expresión del bucle se pueden

usar variables que cambien dentro del propio bucle o que tomen el valor de alguna de las entradas de Arduino que podemos tener asociadas a sensores u otros elementos.

```
while (sensor < 150)  //ejecuta el bucle mientras "sensor" sea menor a 150
{
  ejecuta instrucciones;
}
```

5.2.3 Do... while

El bucle *do... while* funciona de la misma manera que *while*, con la única diferencia de que va a ejecutar al menos 1 vez el bucle, ya que la condición del mismo se comprueba al final.

```
do
{
  sensor = analogRead (1);  //asigna a "sensor" el valor de la entrada analógica 1
}while (sensor < 150)      //repite el bucle mientras "sensor" sea menor que 150
```

5.3 Elementos de control de flujo

El lenguaje de programación de Arduino, como muchos lenguajes de programación, ha heredado características de los primeros lenguajes de programación BASIC, estas características incluyen sentencias para el control del flujo como *Goto*, *Return* y *Break*. Estas sentencias no son muy apreciadas a la hora de programar, ya que en muchas ocasiones rompen el flujo de programa y dificultan la comprensión del mismo. Además en lenguajes de programación de alto nivel como el que usa Arduino se pueden usar funciones y otros elementos alternativos para realizar las mismas tareas, aunque conviene conocer la existencia de estas funciones ya que pueden ser de utilidad en algunasocasiones.

5.3.1 Goto

Esta sentencia realiza un salto a cualquier parte del programa que este marcada con la etiqueta correspondiente, la posición desde la que realicemos el salto quedará almacenada en la pila del programa para que podamos regresar al lugar desde donde realicemos el salto.

```
if (x == 110)
{
  goto marca1;  //salta a la etiqueta "marca1"
}
```

```
marca1:   //etiqueta a la que se saltará (puede estar en cualquier parte)
```

Los saltos que podemos realizar con Arduino están limitados a la pila de programa, no pueden ser infinitos, por lo que no es recomendable anidar bucles y saltos, ya que podemos romper el flujo de programa

5.3.2 Return

Esta sentencia se usa para volver de un salto *goto*. En el momento que es leída por el programa, se cargará la ultima dirección almacenada en la pila de programa, esto hará que se regrese a la posición desde la que se realizó el último salto.

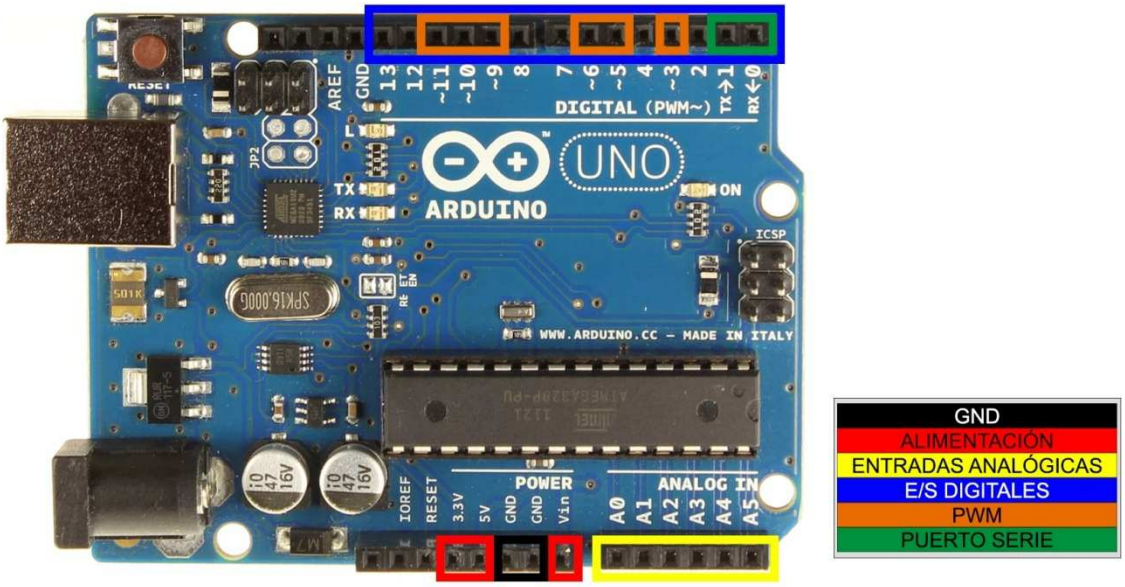
5.3.3 Break

La sentencia break es una sentencia que se debe de evitar a toda costa, tan solo debemos usarla cuando sea totalmente necesario. Esta sentencia rompe la iteración del bucle donde se encuentre, haciendo salir al programa del bucle sin tener en cuenta que se cumplan las condiciones para salir del mismo.

```
for (x = 0; x < 255; x ++)
{
while (sensor < 100)
      {
        x = 0;
        break;   //Rompe el bucle while saliendo al bucle for
      }
}
```

6. Entradas y salidas E/S

Arduino es una plataforma de desarrollo Hardware que cuenta con pines E/S para comunicarse con el exterior. Estos pines E/S tienen características especiales que los hacen propicios para una u otra tarea en función del tipo de pin. Estas E/S pueden englobarse en 3 tipos básicos, E/S analógicas, E/S digitales y E/S para la comunicación serie. Existen también pines de propósito especiales como salidas de reloj u osciladores y pines de alimentación con los que podemos suministrar diferentes tensiones a placas externas



Antes de empezar a trabajar con Arduino, deben de ser configurados los pines de la placa que vayan a ser usados, asignándolos como entradas o como salidas. En ningún caso un mismo pin podrá hacer de entrada y de salida al mismo tiempo.

La configuración de los pines se hará dentro de la función *setup()*, estableciendo el modo de trabajo del pin como entrada o como salida.

La instrucción que se utiliza para realizar la configuración de los pines es ***pinMode***, donde habrá que asignarle el pin que queremos configurar y si queremos que actúe como entrada (INPUT) o como salida (OUTPUT).

```
void setup()
{
  pinMode (10, OUTPUT);//configura el pin 10 como salida
}
```

Los pines de Arduino están configurados por defecto como entradas, por lo que no es necesario indicarles el modo de trabajo si vamos a trabajar con ellos como entradas.

La razón de tener los pines configurados por defecto como entradas, es que las entradas se encuentran en un estado de alta impedancia, lo que va a evitar en muchos casos que dañemos la placa al realizar una mala conexión. Si establecemos un pin como salida y por error entra corriente por dicho pin lo más seguro es que dañemos el microcontrolador de manera irreversible.

Los pines que tengamos configurados como salida (OUTPUT) van a suministrar una corriente máxima de 40mA por separado, sin que la corriente total de las salidas pueda superar los 200mA. Esta corriente es suficiente para hacer brillar un led, pero insuficiente para activar elementos de mayor potencia.

6.1 E/S Digitales

Los pines asignados a E/S digitales, son pines que trabajan con dos estados HIGH (alto) o LOW (BAJO). Según el modelo de Arduino que estemos usando, va a tomar el estado HIGH (alto) como 5v o como 3.3v, el estado LOW (bajo) está asociado a un nivel de voltaje 0.

6.1.1 Lectura de entradas digitales (digitalRead)

Al hacer una lectura digital, vamos a leer el valor de un pin almacenando el resultado como HIGH (alto o 1) o como LOW (bajo o 0).

```
valor = digitalRead (pin);   //la variable "valor" toma el estado asociado al pin
```

Podemos especificar el pin asignándole directamente la numeración del pin digital que queremos leer o con una variable o constante previamente definida.

6.1.2 Escritura de salidas digitales (digitalWrite)

Cuando hacemos una escritura digital vamos a mandar al pin definido previamente como salida el valor HIGH o LOW. El valor HIGH se va a traducir (en función del tipo de Arduino) por una señal de 5 o 3.3 voltios.

```
digitalWrite (pin, HIGH);    //Establece el pin en estado alto (5 o 3.3v)
```

Ejemplo E/S digital

```
#define led 13         //asigna a "led" el valor 13
#define pulsador 7   //asigna a "pulsador" el valor 7
boolean valor;

void setup()
{
  pinMode (led, OUTPUT);   //establece led (pin 13) como salida
  pinMode (pulsador, INPUT);   //establece pulsador (pin 7) como entrada
}
```

```
void loop()
{
  valor = digitalRead (pulsador);   //lee el estado del pulsador y lo asigna a valor
  digitalWrite (led, valor);   //asigna a led el estado de la variable valor
}
```

6.2 E/S Analógicas

Vivimos en un mundo analógico, y en muchas ocasiones, para poder interactuar con el entono no nos es suficiente con detectar o no una señal o poder activar o desactivar cosas, sino que necesitamos cuantificar el valor de magnitudes reales y responder en proporción.

Para poder leer y escribir valores analógicos, Arduino cuenta con una serie de E/S destinadas a este fin, con las que podremos leer y escribir niveles de tensión que irán de 0 a 5 o 3.3v (dependiendo del modelo de Arduino que estemos usando).

6.2.1 Lectura de entradas analógicas (analogRead)

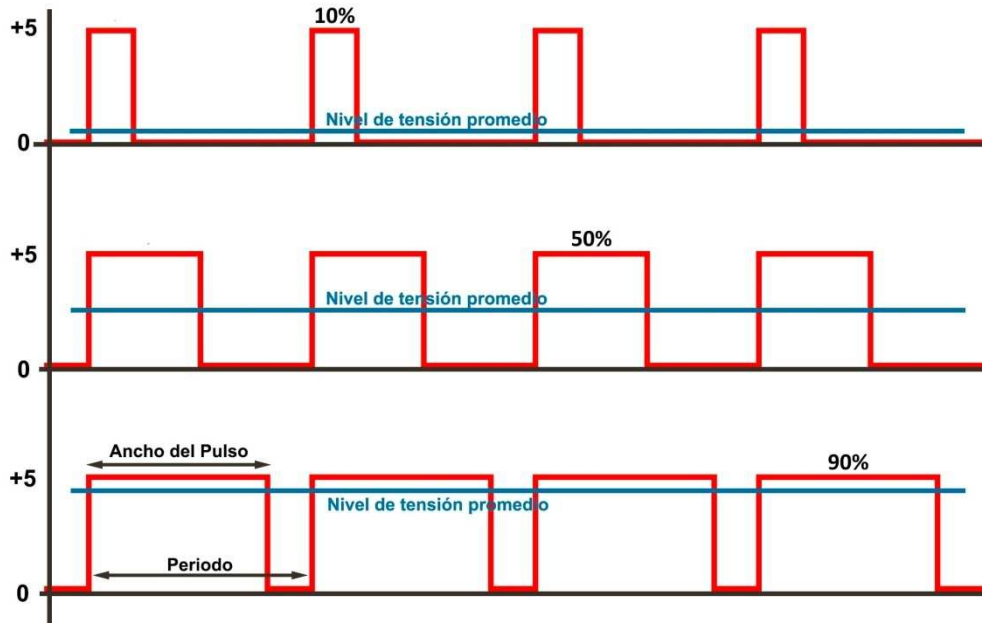
La función *analogRead* realizará una lectura del pin analógico que se le indique, almacenando el valor leído en un registro de 10bit's. Almacenar este valor en un registro de 10 bit's va a implicar que tengamos un rango de valores que va de 0 a 1023, asignándole el 0 a un nivel de 0 voltios y el 1024 a un nivel de 5 voltios, lo que va a determinar la resolución que podemos obtener al hacer lecturas analógicas.

```
valor = analoRead (pin);   //la variable valor toma el nivel del pin analógico
```

Los pines de entrada analógicos no necesitan ser declarados como entrada (INPUT), ya que son siempre entradas.

6.2.2 Escritura de salidas analógicas (analogWrite)

Las salidas analógicas están asociadas a los pines PWM (Pulse Width Modultation) o modulación por ancho de pulso, estos pines, son pines digitales, pero con la particularidad de poseer el PWM, y es que realmente Arduino no entrega a la salida una señal analógica pura, sino que consigue un nivel de tensión determinado por medio de la modulación del ancho de pulso.



El nivel de tensión que tendremos a la salida del pin, va a ser igual al nivel de tensión promedio proporcional al ancho de los pulsos.

Como se observa en la imagen superior, variando en ancho del pulso, podemos obtener una señal promedio equivalente. Para la primera señal, cuyo ancho de pulso es del 10%, nos daría una señal analógica de 0.5v (el 10% de 5V), para la segunda una señal de 2.5v y para la tercera señal obtendríamos una señal equivalente de 4.5v.

Las salidas analógicas trabajan con registros de 8bit's, o lo que es lo mismo, pueden tomar valores comprendidos entre 0 y 255, correspondiendo el 0 a un nivel de 0 voltios y el 255 al máximo voltaje (5 o 3.3 voltios dependiendo de la placa que estemos usando).

```
analogWrite (pin, valor);  //saca por el pin el nivel de tensión equivalente a valor
```

Cuando trabajemos con entradas y salidas analógicas hay que tener en cuenta que al realizar una lectura analógica el resultado se guarda en una variable de 10 bit's, pero la escritura se hace con una variable de 8 bit's, por lo que si queremos usar el valor leído para escribir en algún pin, primero deberemos de adaptar la variable de 10 a 8 bit's, si no hacemos esto, el valor que escribamos va a estar fuera de rango y no va a mostrar el nivel de tensión correcto.

7. Puerto serie

Arduino cuenta con una serie de pines que tienen la capacidad de comunicarse con otros dispositivos usando la comunicación serie para ello. Con esta comunicación serie se puede realizar también una comunicación con el ordenador a través del USB gracias al chip *FDTI* que incorpora la placa y que realiza la conversión USB-serie.

La capacidad de comunicarse con otros dispositivos, hace de Arduino una herramienta muy potente, ya que se pueden crear proyectos muy complejos con multitud de dispositivos que se comuniquen e interactúen entre sí, pero cuando se empieza a programar, es mucho más interesante el poder establecer una comunicación entre nuestro ordenador y Arduino, ya que de esta forma podemos intercambiar información con la placa, pudiendo ver cómo evolucionan las variables del sistema y de esta forma poder detectar posibles problemas de programación.

7.1 Inicialización de la comunicación serie (Serial.begin)

Para poder utilizar el puerto serie hay que inicializarlo estableciendo la velocidad de la conexión. Esta inicialización se hace siempre dentro de la función *setup()* .

Un valor típico para realizar la conexión es 9600 baudios , aunque se pueden asignar otros muchos valores.

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

Según el modelo de Arduino que estemos usando, puede tener 1 o más puertos para la comunicación serie, estos puertos estarán numerados y deberán de abrirse de manera independiente según los que queramos usar.

En el Arduino MEGA se disponen de 4 puertos para la conexión serie, si se desean abrir los 4 puertos el programa quedará de la siguiente manera:

```
void setup()
{
  Serial.begin(9600);
  Serial1.begin(9600);
  Serial2.begin(9600);
  Serial3.begin(9600);
}
```

Debe de tenerse en cuenta que si se inicializa la comunicación serie, los pines asociados al puerto serie que estemos utilizando no podrán ser usados para otro propósito.

7.2 Escritura en el puerto serie (Serial.print)

Si queremos que Arduino muestre información a través del puerto serie, debemos de usar instrucciones que "impriman" en pantalla dicha información.

Para imprimir estos datos se usa el comando *Serial.print*, que mandará a través del puerto serie el dato o la cadena de caracteres que le indiquemos. Esta instrucción tiene algunas variantes, que veremos a continuación.

```
Serial.print(dato, tipo de dato)
```

Esta es la instrucción más común a la hora de enviar datos a través del puerto serie, tan solo hay que indicar el dato que queremos enviar y el formato en el que queremos que muestre dicho dato.

*nota: el tipo de dato es un campo opcional, si no le indicamos ningún tipo, mostrará el dato en formato decimal.

El "tipo de dato" puede tomar los valores BIN (binario), OCT (octal), DEC (decimal) y HEX (hexadecimal). En versiones antiguas de Arduino, también estaba disponible el sacar los datos en formato "BYTE", pero este formato fue eliminado, si queremos sacar un byte en pantalla, podemos utilizar la función *Serial.write(valor)*.

```
Serial.print(78, BIN);   // manda el dato "1001110"
Serial.print(78, OCT);   // manda el dato "116"
Serial.print(78, DEC);   // manda el dato "78"
Serial.print(78, HEX);   // manda el dato "4E"
```

Si estamos trabajando con datos que tienen decimales (*float*) y queremos mostrar un número concreto de ellos, podemos hacerlo poniendo en tipo de dato el número de decimales que queremos mostrar. Por defecto mostrará dos decimales.

```
Serial.println(1.23456, 0);  //manda por el puerto serie el valor "1"
Serial.println(1.23456, 1);  //manda por el puerto serie el valor "1.2"
Serial.println(1.23456, 2);  //manda por el puerto serie el valor "1.23"
Serial.println(1.23456, 3);  //manda por el puerto serie el valor "1.234"
Serial.println(1.23456, 4);  //manda por el puerto serie el valor "1.2346"
```

Además de mostrar datos en pantalla, también es posible mandar cadenas de caracteres, para ello tan solo hay que encerrar el texto que queramos mostrar entre comillas.

```
Serial.print("Hola mundo");   //muestra en pantalla "hola mundo"
```

Cuando mandamos datos a través del puerto serie y queremos visualizar estos datos en pantalla, es recomendable introducir espacios y saltos de línea, ya que si no lo hacemos los datos nos van a aparecer de manera continua y no vamos a poder diferenciar unos de otros. Para ordenar estos datos, podemos introducir tabulaciones o saltos de línea con los siguientes comandos:

```
Serial.print("\t");   //introduce una tabulación entre los datos
Serial.print("\n");   //introduce un salto de línea
```

Si queremos crear datos en líneas diferentes, se puede optar por una variante del *Serial.print* que introduce automáticamente un salto de línea, haciendo que el siguiente dato que se vaya a escribir aparezca en la siguiente línea.

```
Serial.println(dato, tipo de dato)
```

Como en el caso anterior, el tipo de dato será un campo opcional, si no se rellena este campo, el dato aparecerán en formato decimal.

La función *Serial.println* es equivalente a poner:

```
Serial.print (dato, tipo de dato);
Serial.print ("\n");
```

7.3 Lectura del puerto serie (Serial.read)

La instrucción *Serial.read* es una instrucción que va a leer datos entrantes del puerto serie. Estos datos deben de ser almacenados en variables para poder trabajar con ellos.

```
valor = Serial.read();   //almacena el dato del puerto serie en la variable valor
Serial.print(valor);   //"imprime" el dato valor
```

Si hacemos uso de las instrucciones anteriores, vamos a estar constantemente leyendo el puerto serie, pero quizás no haya nada que leer y el dato que estemos almacenando en la variable "valor" sea un dato erróneo. Para evitar estos problemas y optimizar el programa, existe la instrucción *Serial.available()*.

Esta instrucción se usa para comprobar si hay caracteres disponibles para leer en el puerto serie. *Serial.available* va a tomar un valor entero con el número de bytes disponibles para leer que están almacenados en el buffer del puerto serie. Si no hay ningún dato disponible *Serial.avaible* va a valer 0, por lo que es muy fácil el uso de esta función combinada con el condicional *if.*

```
if (Serial.available() > 0)   //si hay algún dato disponible para leer
{
  valor = Serial.read();    //almacena el dato del puerto serie en la variable valor
}
```

Arduino tiene un buffer que puede almacenar como máximo 64 bytes, una vez sobrepasada esta capacidad se empezaran a escribir los datos uno encima de otro, perdiendo la información.

8. Otras instrucciones de interés

Arduino, como otros lenguajes de programación, cuenta con una serie de instrucciones básicas que son muy frecuentes a la hora de programar y que en muchas ocasiones nos ayudan a dar una solución sencilla a los diversos problemas que podamos encontrarnos.

Aquí voy a mostrar algunas de las instrucciones que Arduino trae por defecto, pero no son las únicas, ya que se pueden cargar librerías muy interesantes (como la librería de matemáticas) que añaden funciones con las que podemos realizar tareas complejas de manera muy sencilla.

8.1 Delay(ms)

Esta instrucción detiene la ejecución del programa un tiempo determinado. El tiempo habrá que indicarlo entre paréntesis en milisegundos (1 segundo = 1000 milisegundos).

```
delay (1000);   //espera durante 1 segundo (1000ms)
```

Hay que tener mucho cuidado cuando se usa esta función, ya que se trata de una función bloqueante. Esta función va a detener el flujo de programa, haciendo que durante este tiempo no se detecten eventos como pueden ser presionar un pulsador o la activación de un sensor, esto puede ocasionar graves problemas en la aplicación, por lo que siempre que se pueda hay que evitar usar la instrucción *delay()*.

La instrucción *delay ()* es muy útil cuando ejecutamos tareas que requieren un tiempo mínimo para dicha ejecución, como por ejemplo, para leer memorias externas o comunicarnos con algunos dispositivos vamos a necesitar un tiempo mínimo para acceder a la memoria o establecer la comunicación, este tiempo lo obtenemos deteniendo el programa los milisegundos suficientes para realizar dichas acciones.

8.2 DelayMicroseconds(µs)

La instrucción *delayMicroseconds()* funciona igual que *delay()*, con la diferencia de realizar las temporizaciones usando microsegundos (µs) en lugar de milisegundos (ms).

Esta instrucción nos va a permitir realizar temporizaciones mucho menores, haciendo que el tiempo que detenemos el flojo de programa sea prácticamente imperceptible.

```
delayMicroseconds (10);   //espera durante 0'00001 segundos
```

8.3 Millis ()

La instrucción millis() va a devolver el tiempo en milisegundos trascurridos desde el inicio del programa hasta el momento actual. Esta instrucción es muy interesante, ya que nos permite realizar temporizaciones NO bloqueantes basadas en eventos.

```
tiempo = millis();   //la variable "tiempo" toma el valor del tiempo transcurrido
```

Millis está asociado a un registro de 32bit's, lo que le da la capacidad de temporizar unos 4.294.967.296 milisegundos o lo que es lo mismo 49 días y 17 horas. una vez transcurrido este tiempo el registro se desbordará comenzando la cuenta de nuevo.

Un ejemplo de temporización con millis():

```
#define led 11
#define puls 10
boolean pulsador;
unsigned long tiempo;

void setup()
{
  pinMode(led, OUTPUT);
```

```
  pinMode(puls, INPUT);
}

void loop()
{
  pulsador = digitalRead(puls);
  if(pulsador == HIGH)
  {
  tiempo = millis() + 5000;
  digitalWrite(led, HIGH);
  }

  if(tiempo == millis())
  {
  digitalWrite(led, LOW);
  tiempo = 0;
  }
}
```

8.4 Min(x,y)

La instrucción *min(x, y)* va a comparar dos valores devolviendo el menor de ellos.

```
dato = min(sensor, 100);   //dato toma el valor más pequeño entre "sensor" o 100
```

En el ejemplo anterior "dato" va a tomar el valor del sensor siempre que este valga menos de 100, si es mayor tomará el valor 100. Esa instrucción puede usarse para limitar superiormente el valor de "dato".

8.5 Max (x, y)

La instrucción *max(x, y)* va a comparar dos valores devolviendo el mayor de ellos.

```
dato =max(sensor, 100);
```

Esta instrucción funciona igual que *min(x, y)*, la diferencia es que va a entrega el mayor de los dos números, esto puede ser utilizado para inferiormente el valor que tomará la variable "dato".

ANEXO

CONEXIONADO DE ENTRADAS Y SALIDAS EN ARDUINO

INSTALACIÓN DE ARDUINO Y ENTORNO DE PROGRAMACIÓN

1. Conexionado de entradas y salidas en Arduino

Cuando trabajamos con Arduino debemos tener presente que esta placa no tiene capacidad para controlar elementos de potencia o para gestionar grandes tensiones en los pines de entrada.

Arduino es una plataforma de desarrollo hardware que funciona con 5 o 3.3 voltios (dependiendo de la placa Arduino que usemos) y que tan solo puede entregar 40mA (miliamperios) a las salidas de sus pines de manera independiente sin que se superen los 200mA en conjunto. Debido a estas características, podemos considerar a Arduino como el "cerebro" de nuestro sistema, ya que va a realizar toda la lógica de control pero no será capaz de controlar dispositivos por sí mismo.

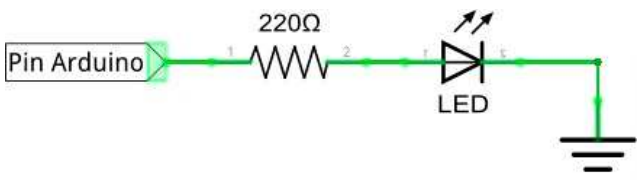
Para realizar el control de estos dispositivos necesitamos "músculos" que entreguen la potencia que Arduino no puede suministrar o que adapten los diferentes niveles de voltaje, para ello ya existen una gran cantidad de shields que dan funcionalidad a Arduino, haciendo posible el control de elementos de potencia, la detección de señales externas y la comunicación con otros dispositivos.

En este bloque se va a tratar el conexionado de elementos a la placa Arduino de forma directa (sin el uso de Shields). El tipo de conexionado que tengamos que realizar, va a depender directamente de la carga conectada.

1.1 Salida de pequeña señal (menor de 40mA)

Arduino puede trabajar con señales de hasta 40mA, esta corriente va a ser suficiente para encender un LED o para establecer la comunicación con la mayoría de dispositivos digitales.

Cuando trabajemos con salidas de señales, tan solo debemos de añadir una resistencia en serie que va a limitar la corriente que sale por el pin. Si el elemento al que vamos a conectar el pin tiene una entrada de alta impedancia, esta resistencia no va a ser necesaria.



El usar una salida digital o analógica (PWM) no va a cambiar nada a la hora de trabajar con pequeñas señales.

1.2 Entrada de pequeña señal

Arduino funciona con niveles de voltaje que van de 0 a 5 o 3.3 voltios (dependiendo de la placa que estemos usando). Por lo que solo va a poder manejar señales que estén dentro de este rango.

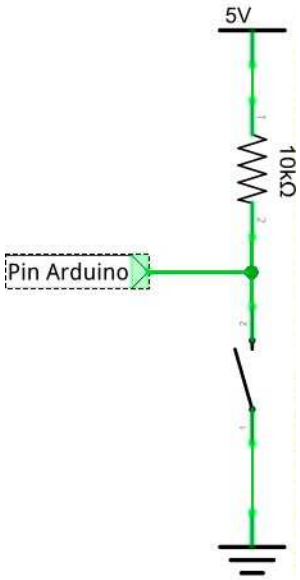
Si la señal con la que estamos trabajando esta dentro del rango que puede soportar Arduino, no vamos a tener ningún problema al cualquier tipo de señal, ya sea digital o analógica.

1.2.1 Entrada digital por interruptor o pulsador

Es muy común en Arduino usar pulsadores o interruptores que conmuten la entrada de los pines entre los niveles ALTO (señal) y BAJO (sin señal) y para ello hay dos maneras posibles de realizar el conexionado, usando resistencias de PullUp o de PullDown.

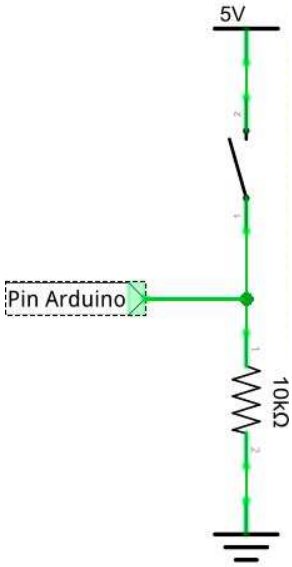
Conexión en PullUp

Al conectar el circuito mediante una resistencia de PullUp, la entrada del pin va a estar a nivel ALTO siempre que el pulsador o interruptor se encuentre abierto. Cuando se sierre el pulsador o interruptor, se va a conectar el pin directamente a GND, cambiando el estado del pin de ALTO a BAJO.



Conexión en PullDown

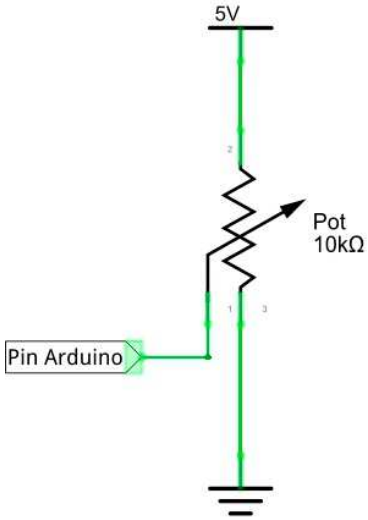
La conexión en PullDown es totalmente opuesta a la conexión en PullUp, esta va a poner el pin en estado BAJO mientras el interruptor o pulsador se encuentren en reposo (abierto), pasando a un estado ALTO al cerrarse el pulsador o interruptor.



El colocar una resistencia de PullUp o PullDown va a hacer que el pin de Arduino esté conectado siempre a un estado lógico ALTO o BAJO y que al cerrar el pulsador o interruptor no se conecten la alimentación y GND directamente, (lo que ocasionaría un cortocircuito) ya que hay una resistencia de 10KΩ entre ambos.

1.2.2 Entrada analógica por potenciómetro.

El uso de un potenciómetro en una de las entradas analógicas, nos va a permitir adaptar el voltaje de entrada en el pin. Dicho nivel de tensión va a ser almacenado en un registro de 10bit's, o lo que es lo mismo, va a valer de 0 a 1023, asignando el valor 0 a un nivel de tensión 0 y 1023 al máximo nivel de tensión de la placa (5 o 3.3voltios).



1.3 Salidas de potencia

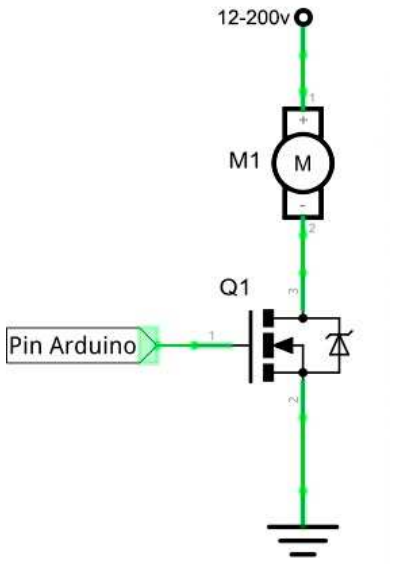
La mayoría de las veces tendremos que controlar elementos que requieren más de los 40mA que es capaz de suministrar cada pin como máximo, para ello debemos de colocar un "driver" que suministre la potencia necesaria con el nivel de tensión adecuado.

En función de la carga que vayamos a colocar y de la regulación de la misma, vamos a optar por un elemento u otro para realizar este control, siendo los elementos más comunes para dicho control los mosfet de potencia, relés y transistores en configuración darlington.

Mosfet de potencia

Los mosfet de potencia se han popularizado mucho en los últimos años, ya que permiten el control de grandes corrientes y velocidades de conmutación muy altas, esto los hace muy adecuados para su uso en fuentes de alimentación y en reguladores de potencia. Otra característica muy importante es que no tienen consumo de corriente en la puerta (se activan por tensión), haciendo que el pin de Arduino no tenga que suministrar corriente.

Los mosfet se comportan como interruptores, cuando se aplica una tensión a la puerta permiten el paso de corriente a través de él.

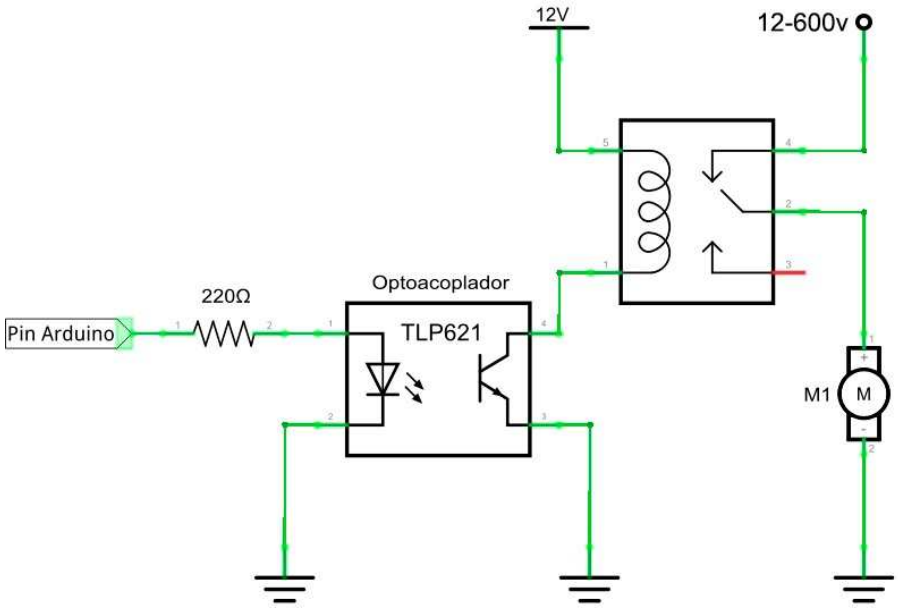


Los mosfet son muy usados junto con los pines PWM, ya que esto va a permitir tener un control de la potencia que se entrega a la carga, pudiendo pudiéndose adaptar a las necesidades de cada momento cambiando el ciclo de trabajo.

Esta regulación es posible gracias a que el mosfet no tiene elementos móviles en su interior, lo que permite tener ciclos de conmutación muy elevados. Esto mismo no sería posible con elementos mecánicos como relés.

Relés

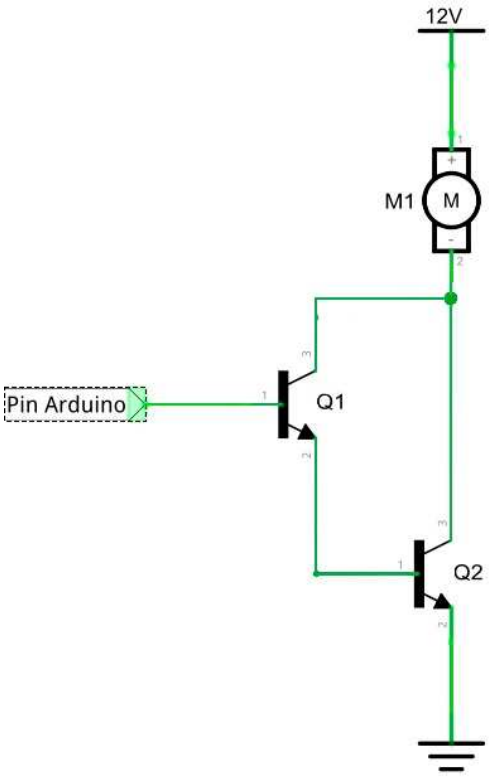
Los relés son interruptores mecánicos activados por corriente. Esta corriente de activación suele superar la corriente que es capaz de suministrar Arduino, además la bobina de relé crea flujos de corriente peligrosos para la placa, por lo que en cualquier caso hay que aislar la activación de la bobina de la placa Arduino usando para ello optoacopladores o transistores en configuración darlington.



Los relés suelen usarse para controlar cargas de todo tipo, ya que es muy amplio el catalogo de relés que existe. La única limitación que tienen los relés frente a los mosfet de potencia es el ciclo de trabajo, los relés no están pensados para conmutar a mucha velocidad, ni para funcionar como reguladores con el PWM. Los relés se utilizan para activar o desactivar cargas de todo tipo siempre que los ciclos de conmutación no sean demasiado elevados.

Transistores en configuración darlington

Los transistores dispuestos en configuración darlington están pensados para suministrar pequeñas potencias. Suelen utilizarse como drivers de relés, para mover pequeños motores o para adaptar diferentes etapas.



Esta configuración se consigue usando 2 transistores bipolares en tándem. Es muy común el uso de integrados que poseen en su interior transistores configurados de esta manera y que ahorran tiempo y espacio a la hora de crear circuitos complejos.

1. Instalación de Arduino y entorno de programación

Antes de empezar a trabajar con nuestro Arduino, será necesario instalar los drivers, ya que nuestro ordenador no va a reconocer la placa cuando la conectemos a alguno de los USB del ordenador. Además de esto también necesitamos el entorno de programación de Arduino, en el que vamos a escribir los programas que posteriormente cargaremos en Arduino.

1.1 Descarga del entorno de desarrollo y drivers

Para empezar hay que ir a la página oficial de Arduino (http://www.arduino.cc) donde podemos descargar las últimas versiones del software. Según nuestro sistema operativo, tendremos que descargar una versión u otra, y dentro de la opción de Windows, podemos hacer la descarga como un archivo ejecutable o como un archivo ZIP.

Dentro del archivo ejecutable, se encuentra tanto el entorno de desarrollo para Arduino, como los driver necesarios para que el dispositivo sea reconocido por nuestro ordenador.

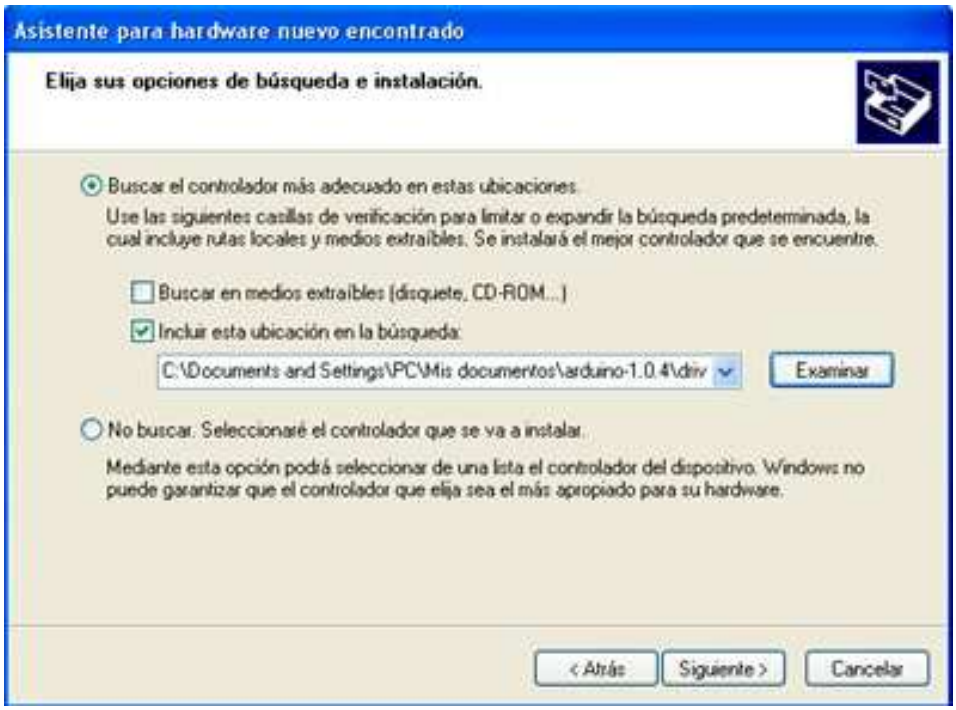
1.2 Instalando los drivers

Para que nuestro ordenador reconozca correctamente la placa Arduino, es necesario instalar los drivers para el procesador FTDI, estos drivers se encuentran dentro de los archivos que hemos descargado, si hemos realizado la descarga en formato ZIP, tendremos que descomprimir la carpeta para acceder a los drivers.

Al conectar Arduino mediante USB al PC, veremos cómo se enciende un LED verde, esto nos indica que la placa está siendo alimentada correctamente. Al momento

de conectar nuestra placa al ordenador, Windows mostrará el mensaje de que hemos conectado un nuevo dispositivo y ejecutara automáticamente el instalador.

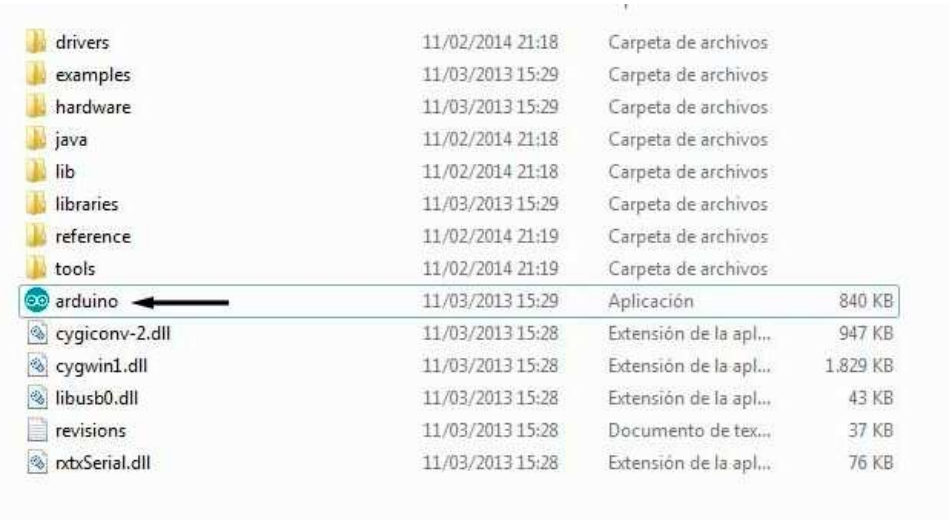
En el asistente de la instalación le diremos que no se conecte a internet para buscar los drivers y en la siguiente pantalla, que los instalaremos desde una localización especifica, que se encontrará dentro de la carpeta que hemos descargado anteriormente, en nuestro caso será la carpeta "drivers" que se encuentra dentro de la carpeta arduino-1.0.5 (la que hemos descargado).



Se deberá de indicar al asistente de instalación la ubicación de la carpeta que hemos descargado, en cuyo interior se encuentran los drivers necesarios para que el ordenador reconozca nuestro Arduino.

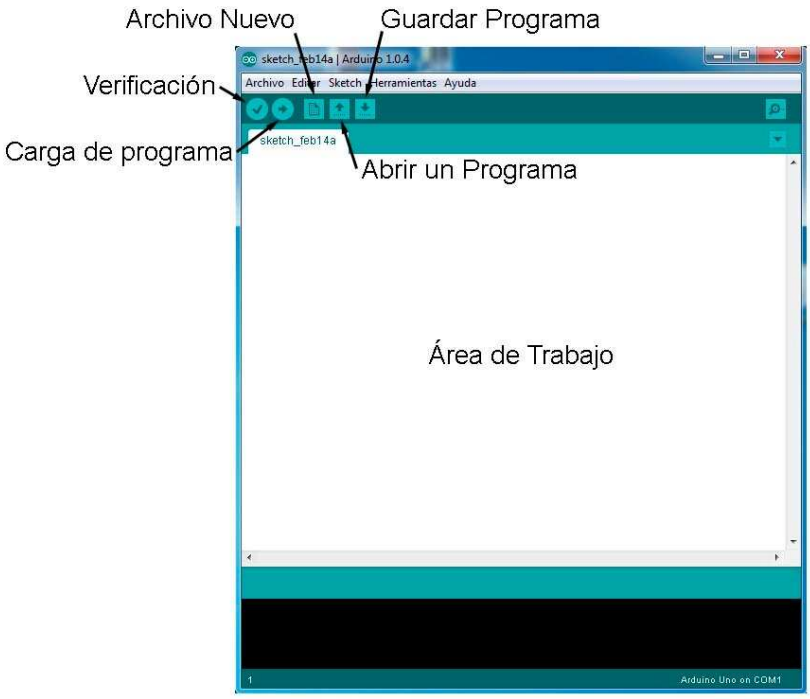
1.3 Entorno de Trabajo

Ahora que nuestro ordenador reconoce perfectamente nuestra placa, es hora de abrir el entorno de trabajo donde programaremos nuestras aplicaciones. El entorno de programación de Arduino no necesita instalación, tan solo tendremos que ir a la carpeta que descargamos y hacer doble clic sobre el icono que pone "arduino".(Recomiendo guardar la carpeta de Arduino en algún lugar de nuestro disco donde creamos que este segura y crear un acceso directo en el escritorio.)



Al hacer esto nos aparecerá la pantalla principal el Sketch, donde podemos diferenciar las siguientes partes:

- Área de trabajo: Esta será el área donde vamos a escribir el programa que queremos ejecutar en Arduino.
- Verificación: Cuando redactemos nuestro programa, podemos verificar que la sintaxis del mismo es correcta y que no hemos cometido ningún error pulsando sobre este icono, si hay algún error, no podremos cargar el programa en Arduino.
- Carga de programa: Cuando tengamos listo nuestro programa y no tenga ningún error, lo cargaremos en Arduino pulsando sobre este botón.
- Archivo nuevo: Abrirá una nueva área de trabajo.
- Abrir un programa: Al pulsar sobre este botón, tendremos la opción de abrir un archivo desde una ubicación especifica o cargar en nuestra área de trabajo una serie de programas o librerías ya creadas y que tiene Arduino por defecto.
- Guardar programa: Guardara en una ubicación especificada por el usuario el contenido del área de trabajo



Cuando tengamos abierto el entorno de trabajo, lo primero que debemos hacer es seleccionar el tipo de placa Arduino que estemos usando y el puerto USB al que la tenemos conectada. Estas opciones las podemos encontrar dentro de la pestaña "herramientas"