
ARDUINO

Índice general

Capítulo 1
Prefacio

Este libro de notas sirve como introducción a la programación y referencia rápida de la estructura de comandos y la sintaxis básica de Arduino. Para mantener la sencillez se han hecho algunas exclusiones que convierten este libro en fuente secundaria de otras webs, libros, talleres y cursos. Esta decisión ha provocado que no se mencione el uso de Arduino sobre placa de prototipado o, por ejemplo, que se excluyan el uso de matrices o formas avanzadas de comunicación serie.
Comenzando con la estructura básica del lenguaje de programación de Arduino, basado en C, este libro de notas continua con una descripción de la sintaxis de los elementos más comunes del lenguaje e ilustra su uso con ejemplos y fragmentos de código. Esto incluye algunas funciones de las librerías base seguidas de un apéndice con esquemas y programas de ejemplo.
Para una introducción al diseño interactivo consultar el "Getting Started with Arduino"de Banzi, también conocido como .ªrduino Booklet". Para los valientes que deseen profundizar en lo intrincado de la programación en C recomiendo "The C Programming Language"de Kernighan y Ritchie, segunda edición, así como Ç in a Nutshell"de Prinz y Crawford, los cuales le darán amplios conocimientos de la sintaxis de programación original. Sobre todo, este libro de notas no habría sido posible sin la gran comunidad de desarrolladores y creadores de material que se encuentra en la web de Arduino, en el Patio de Recreo (Playground) y en el foro de http://www.arduino.cc.

Capítulo 2

Estructura de un programa

La estructura básica del lenguaje de programación de Arduino es bastante simple y se compone de al menos dos partes. Estas dos partes necesarias, o funciones, encierran bloques que contienen declaraciones, estamentos o instrucciones.

```
void setup() //Primera Parte
{
estamentos;
}
void loop() //Segunda Parte
{
estamentos;
}
```

En donde **setup()** es la parte encargada de recoger la configuración y **loop()** es la que contiene el programa que se ejecutará cíclicamente (de ahí el término loop –bucle-). Ambas funciones son necesarias para que el programa trabaje.
La función de configuración (setup) debe contener la declaración de las variables. Es la primera función a ejecutar en el programa, se ejecuta sólo una vez, y se utiliza para configurar o inicializar pinMode (modo de trabajo de las E/S), configuración de la comunicación en serie y otras.
La función bucle (loop) siguiente contiene el código que se ejecutara continuamente (lectura de entradas, activación de salidas, etc) Esta función es el núcleo de todos los programas de Arduino y la que realiza la mayor parte del trabajo.

2.1. setup()

La función setup() se invoca una sola vez cuando el programa empieza. Se utiliza para inicializar los modos de trabajo de los pins, o el puerto serie. Debe ser incluido en un programa aunque no haya declaración que ejecutar. Así mismo se puede utilizar para establecer el estado inicial de las salidas de la placa.

```
void setup()
{
pinMode(pin, OUTPUT); // configura el pin como salida
digitalWrite(pin, HIGH);  // pone el pin en estado HIGH
}
```

2.2. loop()

Después de llamar a setup(), la función loop() hace precisamente lo que sugiere su nombre, se ejecuta de forma cíclica, lo que posibilita que el programa esté respondiendo continuamente ante los eventos que se produzcan en la placa.

```
void loop()
{
digitalWrite(pin, HIGH); // pone en uno (on, 5v) el pin
delay(1000);             // espera un segundo (1000 ms)
digitalWrite(pin, LOW); // pone en cero (off, 0v.) el pin
delay(1000);
}
```

2.3. Funciones

Una función es un bloque de código que tiene un nombre y un conjunto de instrucciones que son ejecutadas cuando se llama a la función. Son funciones setup() y loop() de las que ya se ha hablado. Las funciones de usuario pueden ser escritas para realizar tareas repetitivas y para reducir el tamaño de un programa. Las funciones se declaran asociadas a un tipo de valor "type". Este valor será el que devolverá la función, por ejemplo int se utilizará cuando la función devuelve un dato numérico de tipo entero. Si la función no devuelve

ningún valor entonces se colocará delante la palabra “void”, que significa “función vacía”. Después de declarar el tipo de dato que devuelve la función se debe escribir el nombre de la función y entre paréntesis se escribirán, si es necesario, los parámetros que se deben pasar a la función para que se ejecute.

```
type nombreFuncion(parametros)
{
instruccion;
}
```

La función siguiente devuelve un número entero, delayVal() se utiliza para poner un valor de retraso en un programa que lee una variable analógica de un potenciómetro conectado a una entrada de Arduino. Al principio se declara como una variable local, v recoge el valor leído del potenciómetro que estará comprendido entre 0 y 1023, luego se divide el valor por 4 para ajustarlo a un margen comprendido entre 0 y 255, finalmente se devuelve el valor v y se retornaría al programa principal. Esta función cuando se ejecuta devuelve el valor de tipo entero v.

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

2.4. {} entre llaves

Las llaves sirven para definir el principio y el final de un bloque de instrucciones. Se utilizan para los bloques de programación setup(), loop(), if.., etc.

```
type funcion()
{
instrucciones;
}
```

Una llave de apertura "{“ siempre debe ir seguida de una llave de cierre “}”, si no es así el programa dará errores.

El entorno de programación de Arduino incluye una herramienta de gran utilidad para comprobar el total de llaves. Sólo tienes que hacer click en el punto de inserción de una llave abierta e inmediatamente se marca el correspondiente cierre de ese bloque (llave cerrada).

2.5. ; punto y coma

El punto y coma ";" se utiliza para separar instrucciones en el lenguaje de programación de Arduino. También se utiliza para separar elementos en una instrucción de tipo "bucle for".

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

Nota: Olvidarse de poner fin a una línea con un punto y coma producirá en un error de compilación. El texto de error puede ser obvio, y se referirá a la falta de una coma, o puede que no. Si se produce un error raro y de difícil detección lo primero que debemos hacer es comprobar que los puntos y comas están colocados al final de las instrucciones.

2.6. /*... */ bloque de comentarios

Los bloques de comentarios, o comentarios multi-línea son áreas de texto ignorados por el programa que se utilizan para las descripciones del código o comentarios que ayudan a comprender el programa. Comienzan con / * y terminan con * / y pueden abarcar varias líneas.

```
/* esto es un bloque de comentario no se debe olvidar
cerrar los comentarios estos deben estar equilibrados */
```

Debido a que los comentarios son ignorados por el compilador y no ocupan espacio en la memoria de Arduino pueden ser utilizados con generosidad. También pueden utilizarse para comentar bloques de código con el propósito de anotar informaciones para depuración y hacerlo mas comprensible para cualquiera.
Nota: Dentro de una misma línea de un bloque de comentarios NO se puede escribir otro bloque de comentarios (usando /*..*/).

2.7. // línea de comentarios

Una línea de comentario empieza con // y terminan con la siguiente línea de código. Al igual que los comentarios de bloque, los de línea son ignoradas por el programa y no ocupan espacio en la memoria.

```
// esto es un comentario
```

Una línea de comentario se utiliza a menudo después de una instrucción, para proporcionar más información acerca de lo que hace ésta o para recordarla más adelante.

Capítulo 3
Variables

Una variable es una manera de nombrar y almacenar un valor numérico para su uso posterior por el programa. Como su nombre indica, las variables son números que se pueden variar continuamente en contra de lo que ocurre con las constantes cuyo valor nunca cambia. Una variable debe ser declarada y, opcionalmente, asignarle un valor. El siguiente código de ejemplo declara una variable llamada variableEntrada y luego le asigna el valor obtenido en la entrada analógica del PIN2:

```
int variableEntrada = 0; // declara una variable y le asigna el
    valor 0
variableEntrada = analogRead(2);  //  valor analogico del PIN2
```

'variableEntrada' es la variable en sí. La primera línea declara que será de tipo entero "int". La segunda línea fija a la variable el valor correspondiente a la entrada analógica PIN2. Esto hace que el valor de PIN2 sea accesible en otras partes del código.
Una vez que una variable ha sido asignada, o re-asignada, usted puede probar su valor para ver si cumple ciertas condiciones (instrucciones if..), o puede utilizar directamente su valor. Como ejemplo ilustrativo veamos tres operaciones útiles con variables: el siguiente código prueba si la variable "entradaVariable" es inferior a 100, si es cierto se asigna el valor 100 a "entradaVariable" y, a continuación, establece un retardo (delay) utilizando como valor "entradaVariable" que ahora será como mínimo de valor 100:

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

Nota: Las variables deben tomar nombres descriptivos, para hacer el código más legible. Los nombres de variables pueden ser "contactoSensor" o "pulsador", para ayudar al programador y a cualquier otra persona a leer el código y entender lo que representa la variable. Nombres de variables como "var" o "valor", facilitan muy poco que el código sea inteligible. Una variable puede ser cualquier nombre o palabra que no sea una palabra reservada en el entorno de Arduino.

3.1. Declaración de variables

Todas las variables tienen que declararse antes de que puedan ser utilizadas. Para declarar una variable se comienza por definir su tipo como int (entero), long (largo), float (coma flotante), etc, asignándoles siempre un nombre, y, opcionalmente, un valor inicial. Esto sólo debe hacerse una vez en un programa, pero el valor se puede cambiar en cualquier momento usando aritmética y reasignaciones diversas.
El siguiente ejemplo declara la variable 'entradaVariable' como una variable de tipo entero "int", y asignándole un valor inicial igual a cero. Esto se llama una asignación.

```
int entradaVariable = 0;
```

Una variable puede ser declarada en una serie de lugares del programa y en función del lugar en donde se lleve a cabo la definición esto determinará en que partes del programa se podrá hacer uso de ella.

3.2. Utilización de una variable

Una variable puede ser declarada al inicio del programa antes de la parte de configuración setup(), a nivel local dentro de las funciones, y, a veces, dentro de un bloque, como para los bucles del tipo if.. for.., etc. En función del lugar de declaración de la variable así se determinará el ámbito de aplicación, o la capacidad de ciertas partes de un programa para hacer uso de ella.
Una variable global es aquella que puede ser vista y utilizada por cualquier función y estamento de un programa. Esta variable se declara al comienzo del programa, antes de setup(). Una variable local es aquella que se define dentro de una función o como parte de un bucle. Sólo es visible y sólo puede utilizarse dentro de la función en la que se declaró.

Por lo tanto, es posible tener dos o más variables del mismo nombre en diferentes partes del mismo programa que pueden contener valores diferentes. La garantía de que sólo una función tiene acceso a sus variables dentro del programa simplifica y reduce el potencial de errores de programación.
El siguiente ejemplo muestra cómo declarar a unos tipos diferentes de variables y la visibilidad de cada variable:

```
int value;      // 'value' es visible para cualquier funcion
void setup()
{
// no es necesario configurar nada en este ejemplo
}
void loop()
{
  for (int i=0; i<20;)     // 'i' solo es visible
  {                 // dentro del bucle for
     i++
  }                 // 'f' es visible solo
  float f;               // dentro de loop()
}
```

Capítulo 4
Tipos de datos

4.1. byte

'Byte' almacena un valor numérico de 8 bits sin decimales. Tienen un rango entre 0 y 255.

```
byte unaVariable = 180;      // declara 'unaVariable'
                             // de tipo byte
```

4.2. int

'Int' (Enteros) almacena valores numéricos de 16 bits sin decimales comprendidos en el rango 32767 a -32768.

```
int unaVariable = 1500;      // declara 'unaVariable' como
                             // una variable de tipo entero
```

Nota: Las variables de tipo entero 'int' pueden sobrepasar su valor máximo o mínimo como consecuencia de una operación. Por ejemplo, si x = 32767 y una posterior declaración agrega 1 a x, x = x + 1 entonces el valor se x pasará a ser -32768 (algo así como que el valor da la vuelta).

4.3. long

'Long' se refiere a números enteros (de 32 bits) sin decimales que se encuentran dentro del rango -2147483648 a 2147483647.

```
long unaVariable = 90000;    // declara 'unaVariable'
                             // de tipo long
```

4.4. float

'Float' o ("punto flotante") se aplica a los números con decimales. Los números de punto flotante tienen una mayor resolución que los 'int' (por tratarse de decimales) ocupando también 32 bits con un rango comprendido 3.4028235E +38 a -3.4028235E +38.

```
float unaVariable = 3.14;      // declara 'unaVariable'
                               // de tipo flotante
```

Nota: Los números de punto flotante no son exactos, y pueden producir resultados extraños en las comparaciones. Los cálculos matemáticos de punto flotante son también mucho más lentos que los del tipo de números enteros, por lo que debe evitarse su uso si es posible.

4.5. Arrays

Un array es un conjunto de valores a los que se accede con un número índice. Cualquier valor puede ser recogido haciendo uso del nombre de la matriz y el número del índice. El primer valor de la matriz es el que está indicado con el índice 0. Un array tiene que ser declarado y opcionalmente asignados valores a cada posición antes de ser utilizado.

```
int miArray[] = {valor0, valor1, valor2...}
```

Del mismo modo es posible declarar una matriz indicando el tipo de datos y el tamaño y posteriormente, asignar valores a una posición especifica:

```
int miArray[5];      // declara un array de enteros de 6
                     // posiciones
miArray[3] = 10;       // asigna l valor 10 a la posicion 4
```

Para leer de un array basta con escribir el nombre y la posición a leer:

```
x = miArray[3];      // x ahora es igual a 10 que esta en
                     // la posicion 3 del array
```

Nota: Es conveniente siempre inicializar el array con los valores, o bien definir el tamaño, y no dejarlo sólo con el tipo de datos (int MiArray[];). Las matrices se utilizan a menudo para estamentos de tipo bucle, en los que la variable de incremento del contador del bucle se utiliza como índice o puntero del array. El siguiente ejemplo usa una matriz para el parpadeo de un LED.

Utilizando un bucle tipo for, el contador comienza en cero y escribe el valor que figura en la posición de índice 0 en la serie que hemos escrito dentro del array parpadeo[], en este caso 180, que se envía a la salida analógica tipo PWM (Modulación por ancho de pulso) configurada en el PIN10, se hace una pausa de 200 ms y a continuación se pasa al siguiente valor que asigna el índice "i". Con este ejemplo tendremos un LED parpadeando a diferentes velocidades cada 200ms.

```
int ledPin = 10;          // LED en el PIN 10
byte parpadeo[] = {180, 30, 255, 200, 10, 90, 150, 60};   //
    array de 8 valores
void setup()
{
  pinMode(ledPin, OUTPUT);      // configura la salida
}

void loop()
{
  for(int i=0; i<7; i++)
  {
     analogWrite(ledPin, parpadeo[i]);
     delay(200); // espera 200ms
  }
}
```

Capítulo 5
Aritmética

Los operadores aritméticos que se incluyen en el entorno de programación son: suma, resta, multiplicación y división. Estos devuelven la suma, diferencia, producto o cociente (respectivamente) de dos operandos.

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

La operación se efectúa teniendo en cuenta el tipo de datos que hemos definido para los operandos (int, long, float, etc..), por lo que, por ejemplo, si definimos 9 y 4 como enteros int, 9 / 4 devuelve de resultado 2 en lugar de 2,25 ya que el 9 y 4 son valores de tipo entero int (enteros) y no se ignoran los decimales con este tipo de datos.

Esto también significa que la operación puede sufrir un desbordamiento si el resultado es más grande que lo que puede ser almacenada en el tipo de datos. Recordemos el alcance de los tipos de datos numéricos que ya hemos explicado anteriormente.

Si los operandos son de diferentes tipos, para el cálculo se utilizará el tipo más grande de los operandos en juego. Por ejemplo, si uno de los números (operandos) es del tipo float y otra de tipo integer, para el cálculo se utilizará el método de float es decir, el método de coma flotante.

Elija el tamaño de las variables de tal manera que sea lo suficientemente grande como para que los resultados sean lo precisos que usted desea. Para las operaciones que requieran decimales utilice variables tipo float, pero sea consciente de que las operaciones con este tipo de variables son más lentas a la hora de realizarse el computo.

Nota: Utilice el operador (int) para convertir un tipo de variable a otro (también llamado casting) sobre la marcha. Por ejemplo, i = (int) 3,6 establecerá i igual a 3.

5.1. Asignaciones compuestas

Las asignaciones compuestas combinan una operación aritmética con una variable asignada. Estas son comúnmente utilizadas en los bucles tal como se describe más adelante. Estas asignaciones compuestas pueden ser:

```
x++      // igual que x = x +1, o incremento de x en +1
x--      // igual que x = x - 1, o decremento de x en -1
x += y   // igual que x = x + y, o incremento de x en +y
x -= y   // igual que x = x - y, o decremento de x en -y
x *= y   // igual que x = x * y, o multiplica x por y
x /= y   // igual que x = x / y, o divide x por y
```

Nota: Por ejemplo, x * = 3 hace que x se convierta en el triple del antiguo valor x y por lo tanto x es reasignada al nuevo valor.

5.2. Operadores de comparación

Las comparaciones de una variable o constante con otra se utilizan con frecuencia en las estructuras condicionales del tipo if.. para testear si una condición es verdadera. En los ejemplos que siguen en las próximas páginas se verá su utilización práctica usando los siguientes tipo de condicionales:

```
x == y    // x es igual a y
x != y    // x no es igual a y
x < y     // x es menor que y
x > y     // x es mayor que y
x <= y    // x es menor o igual que y
x >= y    // x es mayor o igual que y
```

5.3. Operadores lógicos

Los operadores lógicos son usualmente una forma de comparar dos expresiones y devolver un VERDADERO o FALSO dependiendo del operador. Existen tres operadores lógicos, AND (&&), OR (||) y NOT (!), que a menudo se utilizan en estamentos de tipo if:
Logica AND:

```
if (x > 0 && x < 5)     // cierto solo si las dos expresiones
                        // son ciertas
```

Logica OR:

```
if (x > 0 || y > 0)     // cierto si una cualquiera de las
                        // expresiones es cierta
```

Logica NOT:

```
if (!x > 0)          // cierto solo si la expresion es
                     // falsa
```

Capítulo 6
Estructuras de control

6.1. if (si condicional)

if es un estamento que se utiliza para probar si una determinada condición se ha alcanzado, como por ejemplo averiguar si un valor analógico está por encima de un cierto número y ejecutar una serie de declaraciones (operaciones) que se escriben dentro de llaves, si es verdad. Si es falso (la condición no se cumple) el programa salta y no ejecuta las operaciones que están dentro de las llaves. El formato para if es el siguiente:

```
if (unaVariable ?? valor)
{
ejecutaInstrucciones;
}
```

En el ejemplo anterior se compara una variable con un valor, el cual puede ser una variable o constante. Si la comparación, o la condición entre paréntesis se cumple (es cierta), las declaraciones dentro de los corchetes se ejecutan. Si no es así, el programa salta sobre ellas y sigue.

Nota: Tenga en cuenta el uso especial del símbolo '=', poner dentro de if (x = 10), podría parecer que es valido pero sin embargo no lo es ya que esa expresión asigna el valor 10 a la variable x, por eso dentro de la estructura if se utilizaría x == 10 que en este caso lo que hace el programa es comprobar si el valor de x es 10. Ambas cosas son distintas por lo tanto dentro de las estructuras if, cuando se pregunte por un valor se debe poner el signo doble de igual ==.

6.2. if... else (si .. sino ..)

if... else viene a ser un estructura que se ejecuta en respuesta a la idea "si esto no se cumple haz esto otro". Por ejemplo, si se desea probar una entrada digital, y hacer una cosa si la entrada fue alto o hacer otra cosa si la entrada es baja, usted escribiría que de esta manera:

```
if (inputPin == HIGH)
{
instruccionesA;
}
else
{
instruccionesB;
}
```

else puede ir precedido de otra condición de manera que se pueden establecer varias estructuras condicionales de tipo unas dentro de las otras (anidamiento) de forma que sean mutuamente excluyentes pudiéndose ejecutar a la vez. Es incluso posible tener un número ilimitado de estos condicionales. Recuerde sin embargo qué sólo un conjunto de declaraciones se llevará a cabo dependiendo de la condición probada:

```
if (inputPin < 500)
  {
  instruccionesA;
  }
else if (inputPin >= 1000)
  {
  instruccionesB;
  {
else
  {
  instruccionesC;
  }
```

Nota: Un estamento de tipo if prueba simplemente si la condición dentro del paréntesis es verdadera o falsa. Esta declaración puede ser cualquier declaración válida. En el anterior ejemplo, si cambiamos y ponemos (inputPin == HIGH). En este caso, el estamento if sólo chequearía si la entrada especificado esta en nivel alto (HIGH), ó +5v.

6.3. For

La declaración for se usa para repetir un bloque de sentencias encerradas entre llaves un número determinado de veces. Cada vez que se ejecutan las instrucciones del bucle se vuelve a testear la condición. La declaración for tiene tres partes separadas por ';' , veamos el ejemplo de su sintaxis:

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

La inicialización de una variable local se produce una sola vez y la condición se testea cada vez que se termina la ejecución de las instrucciones dentro del bucle. Si la condición sigue cumpliéndose, las instrucciones del bucle se vuelven a ejecutar. Cuando la condición no se cumple, el bucle termina.

El siguiente ejemplo inicia el entero i en el 0, y la condición es probar que el valor es inferior a 20 y si es cierto i se incrementa en 1 y se vuelven a ejecutar las instrucciones que hay dentro de las llaves:

```
for (int i=0; i<20; i++)     // declara i y prueba si es
{                  // menor que 20, incrementa i.
digitalWrite(13, HIGH);     // enciende el pin 13
delay(1000);               // espera un seg.
digitalWrite(13, LOW);      // apaga el pin 13
delay(1000);               // espera un seg.
}
```

Nota: El bucle en el lenguaje C es mucho más flexible que otros bucles encontrados en algunos otros lenguajes de programación, incluyendo BASIC. Cualquiera de los tres elementos de cabecera puede omitirse, aunque el punto y coma es obligatorio. También las declaraciones de inicialización, condición y expresión puede ser cualquier estamento válido en lenguaje C sin relación con las variables declaradas. Estos tipos de estados son extraños pero permiten crear soluciones a algunos problemas de programación específicos.

6.4. while

Un bucle del tipo while es un bucle de ejecución continua mientras se cumpla la expresión colocada entre paréntesis en la cabecera del bucle. La

variable de prueba tendrá que cambiar para salir del bucle. La situación podrá cambiar a expensas de una expresión dentro el código del bucle o también por el cambio de un valor en una entrada de un sensor.

```
while (unaVariable ?? valor)
{
ejecutarSentencias;
}
```

El siguiente ejemplo testea si la variable unaVariable es inferior a 200 y si es verdad, ejecuta las declaraciones dentro de los corchetes y continuará ejecutando el bucle hasta que unaVariable no sea inferior a 200.

```
while (unaVariable < 200)     // testea si es menor que 200
{
instrucciones;          // ejecuta las instrucciones
                        // entre llaves
unaVariable++;          // incrementa la variable en 1
}
```

6.5. do... while

El bucle do... while funciona de la misma manera que el bucle while, con la salvedad de que la condición se prueba al final del bucle, por lo que el bucle siempre se ejecutará al menos una vez.

```
do
{
Instrucciones;
} while (unaVariable ?? valor);
```

El siguiente ejemplo asigna el valor leído leeSensor() a la variable x, espera 50 milisegundos y luego continua mientras que el valor de la x sea inferior a 100.

```
do
{
x = leeSensor();
delay(50);
} while (x < 100);
```

Capítulo 7
Entradas-Salidas Digitales

7.1. pinMode(pin, mode)

Esta instrucción es utilizada en la parte de configuración setup () y sirve para configurar el modo de trabajo de un pin pudiendo ser INPUT (entrada) u OUTPUT (salida).

```
pinMode(pin, OUTPUT);      // configura 'pin' como salida
```

Los terminales de Arduino, por defecto, están configurados como entradas, por lo tanto no es necesario definirlos en el caso de que vayan a trabajar como entradas. Los pines configurados como entrada quedan, bajo el punto de vista eléctrico, como entradas en alta impedancia.

Estos pines tienen a nivel interno una resistencia de $20K\Omega$ a las que se puede acceder mediante software. Estas resistencias se accede de la siguiente manera:

```
pinMode(pin, INPUT);           // activa las resistencias internas,
                               // configurando el pin como entrada
digitalWrite(pin, HIGH);       // Pone el pin a 1 (pull-up)
```

Las resistencias internas normalmente se utilizan para conectar las entradas a interruptores. En el ejemplo anterior no se trata de convertir un pin en entrada, es simplemente un método para activar las resistencias internas.

Los pins configurado como OUTPUT (salida) se dice que están en un estado de baja impedancia y pueden proporcionar 40 mA (miliamperios) de corriente a otros dispositivos y circuitos. Esta corriente es suficiente para alimentar un diodo LED (no olvidando poner una resistencia en serie), pero no

es lo suficiente grande como para alimentar cargas de mayor consumo como relés, solenoides o motores.

Un cortocircuito en las patillas Arduino provocará una corriente elevada que puede dañar o destruir el chip ATmega. Puede ser buena idea conectar un pin configurado como salida a un dispositivo externo en serie con una resistencia de 470Ω o de 1000Ω.

7.2. digitalRead(pin)

Lee el valor de un pin digital dando un resultado HIGH (alto) o LOW (bajo). El pin se puede especificar ya sea como una variable o una constante (0-13).

```
valor = digitalRead(pin);      // hace que 'valor' sea igual al
                               // estado leido en 'pin'

digitalWrite(pin, value)
```

Envía al pin definido previamente como OUTPUT el valor HIGH o LOW (poniendo a 1 ó 0 la salida). El pin se puede especificar ya sea como una variable o como una constante (0-13).

```
digitalWrite(pin, HIGH);       // deposita en el 'pin' un valor
                               // HIGH (alto o 1)
```

El siguiente ejemplo lee el estado de un pulsador conectado a una entrada digital y lo escribe en el pin de salida led:

```
int led = 13;          // asigna a LED el valor 13
int boton = 7;         // asigna a boton el valor 7
int valor = 0;         // define el valor y le asigna el
                       // valor 0

void setup()
{
pinMode(led, OUTPUT); // configura el led (pin13) como salida
pinMode(boton, INPUT); // configura boton (pin7) como entrada
}

void loop()
{
valor = digitalRead(boton); //lee el estado de la entrada boton
```

```
digitalWrite(led, valor);    // envia a la salida 'led' el valor
    leido
}
```

Capítulo 8
Entradas-Salidas Analógicas

8.1. analogRead(pin)

Lee el valor de un determinado pin definido como entrada analógica con una resolución de 10 bits. Esta instrucción sólo funciona en los pines (0-5). El rango de valor que podemos leer oscila de 0 a 1023.

```
valor = analogRead(pin); // asigna a 'valor' lo que lee en la
    entrada 'pin'
```

Nota: Los pins analógicos (0-5) a diferencia de los pines digitales, no necesitan ser declarados como INPUT u OUPUT ya que son siempre INPUT.

8.2. analogWrite(pin, value)

Esta instrucción sirve para escribir un pseudo-valor analógico utilizando el procedimiento de modulación por ancho de pulso (PWM) a uno de los pines de Arduino marcados como PWM. El más reciente Arduino, que implementa el chip ATmega368, permite habilitar como salidas analógicas tipo PWM los pines 3, 5, 6, 9, 10 y 11. Los modelos de Arduino más antiguos que implementan el chip ATmega8, sólo tiene habilitadas para esta función los pines 9, 10 y 11. El valor que se puede enviar a estos pines de salida analógica puede darse en forma de variable o constante, pero siempre con un margen de 0-255.

```
analogWrite(pin, valor);     // escribe 'valor' en el 'pin'
                             // definido como analogico
```

Si enviamos el valor 0 genera una salida de 0 voltios en el pin especificado; un valor de 255 genera una salida de 5 voltios de salida en el pin especificado. Para valores de entre 0 y 255, el pin saca tensiones entre 0 y 5 voltios - cuanto mayor sea el valor, más a menudo estará a HIGH (5 voltios). Teniendo en cuenta el concepto de señal PWM , por ejemplo, un valor de 64 equivaldrá a mantener 0 voltios de tres cuartas partes del tiempo y 5 voltios a una cuarta parte del tiempo; un valor de 128 equivaldrá a mantener la salida en 0 la mitad del tiempo y 5 voltios la otra mitad del tiempo, y un valor de 192 equivaldrá a mantener en la salida 0 voltios una cuarta parte del tiempo y de 5 voltios de tres cuartas partes del tiempo restante.

Debido a que esta es una función de hardware, en el pin de salida analógica (PWM) se generará una onda constante después de ejecutada la instrucción analogWrite hasta que se llegue a ejecutar otra instrucción analogWrite (o una llamada a digitalRead o digitalWrite en el mismo pin).

Nota: Las salidas analógicas a diferencia de las digitales, no necesitan ser declaradas como INPUT u OUTPUT.

El siguiente ejemplo lee un valor analógico de un pin de entrada analógica, convierte el valor dividiéndolo por 4 y envía el nuevo valor convertido a una salida del tipo PWM o salida analógica:

```
int led = 10;        // define el pin 10 como 'led'
int analog = 0;      // define el pin 0 como 'analog'
int valor;           // define la variable 'valor'

void setup(){}        // no es necesario configurar
                      // entradas y salidas

void loop()
{
valor = analogRead(analog);       // lee el pin 0 y lo asocia a
                                  // la variable valor
valor /= 4;                       // divide valor entre 4 y lo
                                  // reasigna a valor
analogWrite(led, value);          // escribe en el pin 10 'valor'
}
```

Capítulo 9
Tiempo

9.1. delay(ms)

Detiene la ejecución del programa la cantidad de tiempo en milisegundos que se indica en la propia instrucción. De tal manera que 1000 equivale a 1 segundo.

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

9.2. millis()

Devuelve el número de milisegundos transcurrido desde el inicio del programa en Arduino hasta el momento actual. Normalmente será un valor grande (dependiendo del tiempo que este en marcha la aplicación después de cargada o después de la última vez que se pulsó el botón reset de la tarjeta).

```
valor = millis();      // valor recoge el numero de
                       // milisegundos
```

Nota: Este número se desbordará (si no se resetea de nuevo a cero), después de aproximadamente 49 días.

Capítulo 10
Matemática

10.1. min(x, y)

Calcula el mínimo de dos números para cualquier tipo de datos devolviendo el menor de ellos.

```
valor = min(valor, 100); // asigna a 'valor' el minimo
                         // de los dos numeros especificados.
```

Si valor es menor que 100, valor recogerá su propio valor. Si valor es mayor que 100, valor pasará a valer 100.

10.2. max(x, y)

Calcula el máximo de dos números para cualquier tipo de datos devolviendo el número mayor ellos.

```
valor = max(valor, 100); // asigna a 'valor' el mayor de
                         // los dos numeros 'valor' y 100.
```

De esta manera nos aseguramos de que valor será como mínimo 100.

Capítulo 11
Números aleatorios

11.1. randomSeed(seed)

Establece un valor, o semilla, como punto de partida para la función random().

```
randomSeed(valor);      // hace que 'valor' sea la semilla del
                        // random
```

Debido a que Arduino es incapaz de crear un verdadero número aleatorio, randomSeed le permite colocar una variable, constante, u otra función de control dentro de la función random, lo que permite generar números aleatorios al azar. Hay una variedad de semillas, o funciones, que pueden ser utilizados en esta función, incluido millis() o incluso analogRead() que permite leer ruido eléctrico a través de un pin analógico.

11.2. random(max)

Devuelve un valor aleatorio entre 0 y max.

```
numAleatorio = random(300); // escribe un numero aleatorio de 0
    a 300 en la variable 'numAleatorio'
```

11.3. random(min, max)

La función random devuelve un número aleatorio entero de un intervalo de valores especificado entre los valores min y max.

```
valor = random(100, 200);     // asigna a la variable 'valor' un
    numero aleatorio comprendido entre 100 y 200
```

Nota: Use esta función después de usar el randomSeed(). El siguiente ejemplo genera un valor aleatorio entre 0 y 255 y lo envía a una salida analógica PWM :

```
int randNumber;    // variable que almacena el valor aleatorio
int led = 10;      // define led como 10

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

void loop() {
randomSeed(millis());     // genera una semilla para aleatorio a
    partir de la funcion millis()
randNumber = random(255);     // genera numero aleatorio entre 0 y
     255
analogWrite(led, randNumber); // envia a la salida led de tipo
   PWM el valor
delay(500);       // espera 0,5 seg.
}
```

Capítulo 12
Serial

12.1. Serial.begin(rate)

Abre el puerto serie y fija la velocidad en baudios para la transmisión de datos en serie. El valor típico de velocidad para comunicarse con el ordenador es 9600, aunque otras velocidades pueden ser soportadas.

```
void setup()

{
Serial.begin(9600);     // abre el Puerto serie
}    // configurando la velocidad en 9600 bps
```

Nota: Cuando se utiliza la comunicación serie los pines digitales 0 (RX) y 1 (TX) no pueden utilizarse para otros propósitos.

12.2. Serial.println(data)

Imprime los datos en el puerto serie, seguido por un retorno de carro y salto de línea. Este comando toma la misma forma que Serial.print (), pero es más fácil para la lectura de los datos en el Monitor Serie del software.

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

Nota: Para obtener más información sobre las distintas posibilidades de Serial.println() y Serial.print() puede consultarse el sitio web de Arduino.

El siguiente ejemplo toma de una lectura analógica del pin 0 y envía estos datos al ordenador cada segundo.

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

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

12.3. Serial.println(data, data type)

Vuelca o envía un número o una cadena de caracteres al puerto serie, seguido de un caracter de retorno de carro 'CR' (ASCII 13, or '\r')y un caracter de salto de línea 'LF'(ASCII 10, or '\n'). Toma la misma forma que el comando Serial.print()

- Serial.println(b) vuelca o envía el valor de b como un número decimal en caracteres ASCII seguido de 'CR' y 'LF'.
- Serial.println(b, DEC) vuelca o envía el valor de b como un número decimal en caracteres ASCII seguido de 'CR' y 'LF'.
- Serial.println(b, HEX) vuelca o envía el valor de b como un número hexdecimal en caracteres ASCII seguido de 'CR' y 'LF'.
- Serial.println(b, OCT) vuelca o envía el valor de b como un número octal en caracteres ASCII seguido de 'CR' y 'LF'.
- Serial.println(b, BIN) vuelca o envía el valor de b como un número binario en caracteres ASCII seguido de 'CR' y 'LF'.
- Serial.print(b, BYTE) vuelca o envía el valor de b como un byteseguido de 'CR' y 'LF'.
- Serial.println(str) vuelca o envía la cadena de caracteres como una cadena ASCII seguido de 'CR' y 'LF'.
- Serial.println() sólo vuelca o envía 'CR' y 'LF'.

12.4. Serial.print(data, data type)

Vuelca o envía un número o una cadena de carateres, al puerto serie. Dicho comando puede tomar diferentes formas, dependiendo de los parámetros que utilicemos para definir el formato de volcado de los números.

12.4.1. Parámetros

- data: el número o la cadena de caracteres a volcar o enviar.
- data type: determina el formato de salida de los valores numéricos (decimal, octal, binario, etc...) DEC, OCT, BIN, HEX, BYTE.

12.4.2. Ejemplos

Serial.print(b) Vuelca o envía el valor de b como un número decimal en caracteres ASCII.

```
int b = 79; Serial.print(b); // envia "79".
```

Serial.print(b, DEC) Vuelca o envía el valor de b como un número decimal en caracteres ASCII.

```
int b = 79;
Serial.print(b, DEC); // envia "79".
```

Serial.print(b, HEX) Vuelca o envía el valor de b como un número hexdecimal en caracteres ASCII.

```
int b = 79;
Serial.print(b, HEX); // envia "4F".
```

Serial.print(b, OCT) Vuelca o envía el valor de b como un número octal en caracteres ASCII.

```
int b = 79;
Serial.print(b, OCT); // envia "117".
```

Serial.print(b, BIN) Vuelca o envía el valor de b como un número binario en caracteres ASCII.

```
int b = 79;
Serial.print(b, BIN); // envia "1001111".
```

Serial.print(b, BYTE) Vuelca o envía el valor de b como un byte.

```
int b = 79;

Serial.print(b, BYTE); // Devuelve el caracter 'O', el cual
    representa el caracter ASCII del valor 79. (Ver tabla ASCII )
    .
```

Serial.print(str) Vuelca o envía la cadena de caracteres como una cadena ASCII.

```
Serial.print("Hello World!"); // envia "Hello World!".
```

12.5. Serial.avaible()

```
int Serial.available()
```

Devuelve un entero con el número de bytes (carácteres) disponibles para leer desde el buffer serie, ó 0 si no hay ninguno. Si hay algún dato disponible, SerialAvailable() será mayor que 0. El buffer serie puede almacenar como máximo 128 bytes.
Ejemplo:

```
int incomingByte = 0; // almacena el dato serie
void setup() {
Serial.begin(9600); // abre el puerto serie, y le asigna la
    velocidad de 9600 bps
}
void loop() {
// envia datos solo si los recibe:
if (Serial.available() > 0) {
// lee el byte de entrada:
incomingByte = Serial.read();
//lo vuelca a pantalla
```

```
Serial.print("He recibido: "); Serial.println(incomingByte, DEC);
}
}
```

12.6. Serial.Read()

```
int Serial.Read()
```

Lee o captura un byte (carácter) desde el puerto serie. Devuelve :El siguiente byte (carácter) desde el puerto serie, ó -1 si no hay ninguno. Ejemplo

```
int incomingByte = 0; // almacenar el dato serie
void setup() {
Serial.begin(9600); // abre el puerto serie,y le asigna la
    velocidad de 9600 bps
}
void loop() {
// envia datos solo si los recibe:
if (Serial.available() > 0) {
// lee el byte de entrada:
incomingByte = Serial.read();
//lo vuelca a pantalla
Serial.print("He recibido: "); Serial.println(incomingByte, DEC);
}
}
```

Appendices

Apéndice A
Formas de Conexionado de entradas y salidas

A.1. Salida digital

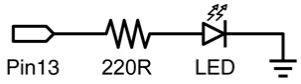

Figura A.1: Conexion Salida digital

Éste es el ejemplo básico equivalente al 'hola mundo' de cualquier lenguaje de programación haciendo simplemente el encendido y apagado de un led. En este ejemplo el LED está conectado en el pin13, y se enciende y se apaga 'parpadea' cada segundo. La resistencia que se debe colocar en serie con el led en este caso puede omitirse ya que el pin13 de Arduino ya incluye en la tarjeta esta resistencia,

```
int ledPin = 13; // LED en el pin digital 13

void setup()      // configura el pin de salida
{
   pinMode(ledPin, OUTPUT); // configura el pin 13 como salida
}

void loop()       // inicia el bucle del programa
{
   digitalWrite(ledPin, HIGH); // activa el LED
   delay(1000); // espera 1 segundo
   digitalWrite(ledPin, LOW); // desactiva el LED
   delay(1000); // espera 1 segundo
}
```

A.2. Entrada digital

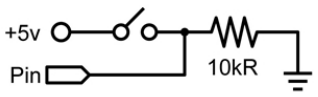

Figura A.2: Conexión Entrada digital

Ésta es la forma más sencilla de entrada con sólo dos posibles estados: encendido o apagado. En este ejemplo se lee un simple switch o pulsador conectado a PIN2. Cuando el interruptor está cerrado en el pin de entrada se lee ALTO y encenderá un LED colocado en el PIN13:

```
int ledPin = 13; // pin 13 asignado para el LED de salida
int inPin = 2; // pin 2 asignado para el pulsador

void setup() // Configura entradas y salidas
{
   pinMode(ledPin, OUTPUT); // declara LED como salida
   pinMode(inPin, INPUT); // declara pulsador como entrada
}

void loop()
{
   if (digitalRead(inPin) == HIGH) // testea si la entrada esta
       activa HIGH
   {
      digitalWrite(ledPin, HIGH); // enciende el LED
      delay(1000); // espera 1 segundo
      digitalWrite(ledPin, LOW); // apaga el LED
   }
}
```

A.3. Salida de alta corriente de consumo

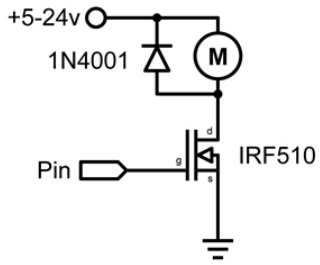

Figura A.3: Conexión salida de alta corriente de consumo

A veces es necesario controlar cargas de más de los 40 mA que es capaz de suministrar la tarjeta Arduino. En este caso se hace uso de un transistor MOSFET que puede alimentar cargas de mayor consumo de corriente. El siguiente ejemplo muestra como el transistor MOSFET conmuta 5 veces cada segundo.

Nota: El esquema muestra un motor con un diodo de protección por ser una carga inductiva. En los casos que las cargas no sean inductivas no será necesario colocar el diodo.

```
int outPin = 5; // pin de salida para el MOSFET

void setup()
{
   pinMode(outPin, OUTPUT); // pin5 como salida
}

void loop()
{
   for (int i=0; i<=5; i++)      // repetir bucle 5 veces
   {
      digitalWrite(outPin, HIGH); // activa el MOSFET
      delay(250); // espera 1/4 segundo
      digitalWrite(outPin, LOW); // desactiva el MOSFET
      delay(250); // espera 1/4 segundo
   }
   delay(1000); // espera 1 segundo
}
```

A.4. Salida analógica del tipo pwm

PWM (modulación de impulsos en frecuencia)

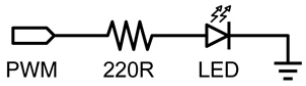

Figura A.4: Conexión salida analógica del tipo pwm

La Modulación por Anchura de Pulsos (PWM) es una forma de conseguir una "falsa" salida analógica. Esto podría ser utilizado para modificar el brillo de un LED o controlar un servo motor. El siguiente ejemplo hace que el LED se ilumine y se apague lentamente haciendo uso de dos bucles.

```
int ledPin = 9; // pin PWM para el LED

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

void loop()
{
   for (int i=0; i<=255; i++) // el valor de i asciende
   {
      analogWrite(ledPin, i); // se escribe el valor de I en el
         PIN de salida del LED
      delay(100); // pauses for 100ms
   }
   for (int i=255; i>=0; i--) // el valor de I desciendei
   {
      analogWrite(ledPin, i); // se escribe el valor de ii
      delay(100); // pasusa durante 100ms
   }
}
```

A.5. Entrada con potenciómetro

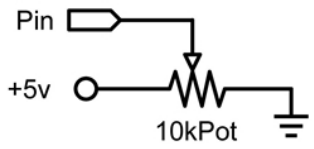

Figura A.5: Conexión entrada con potenciometro

El uso de un potenciómetro y uno de los pines de entrada analógica-digital de Arduino (ADC) permite leer valores analógicos que se convertirán en valores dentro del rango de 0-1023. El siguiente ejemplo utiliza un potenciómetro para controlar un el tiempo de parpadeo de un LED.

```
int potPin = 0; // pin entrada para potenciometro
int ledPin = 13;      // pin de salida para el LED

void setup()
{
   pinMode(ledPin, OUTPUT);      // declara ledPin como SALIDA
}

void loop()
{
   digitalWrite(ledPin, HIGH); // pone ledPin en on
   delay(analogRead(potPin)); // detiene la ejecucion un tiempo '
       potPin'
   digitalWrite(ledPin, LOW); // pone ledPin en off
   delay(analogRead(potPin)); // detiene la ejecucion un tiempo '
       potPin'
}
```

A.6. Entrada conectada a resistencia variable

Figura A.6: Conexión entrada con resistencia variable

Las resistencias variables como los sensores de luz LCD, los termistores, sensores de esfuerzos, etc, se conectan a las entradas analógicas para recoger valores de parámetros físicos. Este ejemplo hace uso de una función para leer el valor analógico y establecer un tiempo de retardo. Este tiempo controla el brillo de un diodo LED conectado en la salida.

```
int ledPin = 9; // Salida analogica PWM para conectar a LED
int analogPin = 0; // resistencia variable conectada a la entrada
    analogica pin 0

void setup(){}    // no es necesario configurar entradas y
   salidas

void loop()
{
   for (int i=0; i<=255; i++) // incremento de valor de i
   {
      analogWrite(ledPin, i); // configura el nivel brillo con el
          valor de i
      delay(delayVal()); // espera un tiempo en milisegundos
   }
   for (int i=255; i>=0; i--) // decrementa el valor de i
   {
      analogWrite(ledPin, i); // configura el nivel de brillo con
          el valor de i
      delay(delayVal()); // espera un tiempo en milisegundos
   }
}

int delayVal() // Metodo para recoger el tiempo de retardo
{
   int v; // crea una variable temporal (local)

   v    = analogRead(analogPin); // lee valor analogico
   v *= 2;     // convierte el valor leido de 0-1023 a 0-2046
   return v; // devuelve el valor v
}
```

A.7. Salida conectada a servo

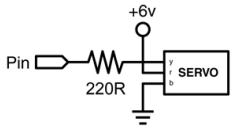

Figura A.7: Conexión salida conectada a servo

Los servos de los juguetes tienen un tipo de motor que se puede mover en un arco de 180 º y contienen la electrónica necesaria para ello. Todo lo que se necesita es un pulso enviado cada 20ms. Este ejemplo utiliza la función servoPulse para mover el servo de 10º a 170 º.

```
int servoPin = 2; // servo conectado al pin digital 2
int myAngle; // angulo del servo de 0-180
int pulseWidth; // anchura del pulso para la funcion servoPulse

void setup()
{
   pinMode(servoPin, OUTPUT); // configura pin 2 como salida
}

void servoPulse(int servoPin, int myAngle)
{
   pulseWidth = (myAngle * 10) + 600; // determina retardo
   digitalWrite(servoPin, HIGH); // activa el servo
   delayMicroseconds(pulseWidth); // pausa
   digitalWrite(servoPin, LOW); // desactiva el servo
   delay(20); // retardo de refresco
}

void loop()
{
// el servo inicia su recorrido en 10 y gira hasta 170
   for (myAngle=10; myAngle<=170; myAngle++)
   {
      servoPulse(servoPin, myAngle);
   }
// el servo vuelve desde 170 hasta 10
   for (myAngle=170; myAngle>=10; myAngle--)
   {
      servoPulse(servoPin, myAngle);
   }
}
```

Apéndice B

Cómo escribir una librería para Arduino

Este documento explica cómo crear una librería para Arduino. Se comienza con un programa que realiza, mediante encendido y apagado de un led, el código morse y se explica cómo convertir este en una función de librería. Esto permite a otras personas utilizar fácilmente el código que has escrito cargándolo de una forma sencilla.

Se comienza con el programa de un sencillo código Morse: La palabra a generar es SOS (. . . - - - . . .).

```
// Genera SOS en codigo Morse luminoso

int pin = 13;

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

void loop() //Programa principal que genera '. . .', '- - -' y '.
    . . .'
{
   dot(); dot(); dot(); //Genera la S (. . .)
   dash(); dash(); dash(); // Genera la O (- - -)
   dot(); dot(); dot(); // Genera la S (. . .)
   delay(3000); //Espera un tiempo
}

void dot() //Procedimiento para generar un punto
{
   digitalWrite(pin, HIGH);
   delay(250);
   digitalWrite(pin, LOW);
```

```
    delay(250);
}

void dash()       //Procedimiento para generar una raya
{
    digitalWrite(pin, HIGH);
    delay(1000);
    digitalWrite(pin, LOW);
    delay(250);
}
```

Si se ejecuta este programa, se ejecuta el código SOS (llamada de solicitud de auxilio) en el pin 13.
El programa tiene distintas partes que tendremos que poner en nuestra librería. En primer lugar, por supuesto, tenemos las funciones dot() (punto) y dash() (raya) que se encargar de que el led parpadee de manera corta o larga respectivamente. En segundo lugar, tenemos la línea ledPin que utilizamos para determinar el pin a utilizar. Por último, está la llamada a la función pinMode() que inicializa el pin como salida.
Vamos a empezar a convertir el programa en una librería.
Necesitas por lo menos dos archivos en una librería: un archivo de cabecera (con la extensión .h) y el archivo fuente (con la extensión .cpp). El fichero de cabecera tiene definiciones para la librería: básicamente una lista de todo lo que contiene, mientras que el archivo fuente tiene el código real. Vamos a llamar a nuestra biblioteca 'Morse', por lo que nuestro fichero de cabecera se Morse.h. Echemos un vistazo a lo que sucede en ella. Puede parecer un poco extraño al principio, pero lo entenderá una vez que vea el archivo de origen que va con ella.
El núcleo del archivo de cabecera consiste en una línea para cada función en la biblioteca, envuelto en una clase junto con las variables que usted necesita:

```
class Morse
{
    public: Morse(int pin);
    void dot();
    void dash();
private:
    int _pin;
};
```

Una clase es simplemente una colección de funciones y variables que se mantienen unidos todos en un solo lugar. Estas funciones y variables pueden ser

públicos, lo que significa que puede ser utilizadas por quienes utilizan la librería, o privadas, lo que significa que sólo se puede acceder desde dentro de la propia clase. Cada clase tiene una función especial conocida como un constructor, que se utiliza para crear una instancia de la clase. El constructor tiene el mismo nombre que la clase, y no devuelve nada.
Usted necesita dos cosas más en el fichero de cabecera. Uno de ellos es un #include, declaración que le da acceso a los tipos estándar y las constantes del lenguaje de Arduino (esto se añade automáticamente en todos los programas que hacemos con Arduino, pero no a las librerías), por lo que debemos incluirlas (poniéndolas por encima de la definición de clase dada anteriormente):

```
#include "WConstants.h"
```

Por último, se colocara delante del código la cabecera siguiente:

```
#ifndef Morse_h
#define Morse_h
// el estamento #include y el resto del codigo va aqui..
#endif
```

Básicamente, esto evita problemas si alguien accidentalmente pone #include en la librería dos veces.
Por último, por lo general, se pone un comentario en la parte superior de la librería con su nombre, una breve descripción de lo que hace, quien la escribió, la fecha y la licencia. Echemos un vistazo a la cabecera completa disposición del fichero de cabecera h:

Fichero Morse.h

```
/*
Morse.h - Library for flashing Morse code.
Created by David A. Mellis, November 2, 2007. Released into the
    public domain.
*/
#ifndef Morse_h
#define Morse_h
#include "WConstants.h"

class Morse
{
    public: Morse(int pin); void dot();
    void dash();
    private:
```

```
    int _pin;
};

#endif
```

Ahora vamos a escribir las diversas partes del archivo fuente de la librería, Morse.cpp. Primero se ponen un par de declaraciones mediante #include. Estas incluyen resto del código de acceso a las funciones estándar de Arduino, ya que en las definiciones figuran en el archivo de cabecera:

```
#include "WProgram.h"
#include "Morse.h"
```

Luego viene el constructor. Ahora se indicará lo que debería suceder cuando alguien crea una instancia a la clase. En este caso, el usuario especifica el pin que les gustaría utilizar. Configuramos el pin como salida guardarlo en una variable privada para su uso en las otras funciones:

```
Morse::Morse(int pin)
{
    pinMode(pin, OUTPUT);
    _pin = pin;
}
```

Hay un par de cosas extrañas en este código. El primero es el Morse:: antes del nombre de la función. Esto indica que la función es parte de la clase Morse. Verá este de nuevo en las otras funciones en la clase. La segunda cosa inusual es el guión bajo en el nombre de nuestra variable privada, _pin. Esta variable puede tener cualquier nombre que desee, siempre y cuando coincida con la definición que figura en el fichero de cabecera. La adición de un guión bajo al comienzo del nombre es una convención para dejar claro que las variables son privadas, y también a distinguir el nombre de la del argumento a la función (pin en este caso).
Después viene el código del programa que queremos convertir en una función (¡por fin!). Parece casi igual, excepto con Morse:: delante de los nombres de las funciones, y _pin en lugar de pin:

```
void Morse::dot()
{
    digitalWrite(_pin, HIGH); delay(250);
    digitalWrite(_pin, LOW); delay(250);
}
```

```
void Morse::dash()
{
  digitalWrite(_pin, HIGH); delay(1000);
  digitalWrite(_pin, LOW); delay(250);
}
```

Por último, es típico incluir el comentario de cabecera en la parte superior de la fuente así como el archivo. Vamos a ver el fichero completo:
Fichero Morse.cpp

```
/*
Morse.cpp - Library for flashing Morse code. Created by David A.
    Mellis, November 2, 2007. Released into the public domain.
*/

#include "WProgram.h"
#include "Morse.h"

Morse::Morse(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}

void Morse::dot()
{
  digitalWrite(_pin, HIGH); delay(250);
  digitalWrite(_pin, LOW); delay(250);
}

void Morse::dash()
{
  digitalWrite(_pin, HIGH); delay(1000);
  digitalWrite(_pin, LOW); delay(250);
}
```

Y eso es todo lo que necesita (hay algunas otras cosas opcionales, pero vamos a hablar de eso más adelante).
Ahora vamos a ver cómo se utiliza la librería.
En primer lugar, debemos crear una carpeta llamada Morse dentro del subdirectorio libraries de la aplicación Arduino. Copiar o mover los archivos Morse.h y Morse.cpp en esa carpeta. Ahora lanzar la aplicación Arduino. Si abres el menú Sketch>Import Library, deberás ver el objeto Morse. Si realiazs

alguna modificación en la libería, deberás reiniciar el entorno de Arduino para así recompilarla. Si aparece algún fallo relacionado con la compilación de la biblioteca, asegúrese de que están realmente los archivos .cpp y .h (sin extensión adicional .pde o .txt, por ejemplo).
Veamos como podemos escribir nuestro nuevo programa SOS haciendo uso de la nueva librería:

Programa para Arduino

```
#include <Morse.h>

Morse morse(13);

void setup() {}

void loop()
{
    morse.dot(); morse.dot(); morse.dot();
    morse.dash(); morse.dash(); morse.dash();
    morse.dot(); morse.dot(); morse.dot();
    delay(3000);
}
```

Hay algunas diferencias con respecto al antiguo programa (además del hecho de que algunos de los códigos se han incorporado a la librería).
En primer lugar, hemos añadido un estamento #include en la parte superior del programa. Esto hace que la librería Morse quede a disposición del programa y la incluye en el código. Esto significa que ya no necesitan una librería en el programa, usted debe borrar el #include para ahorrar espacio.
En segundo lugar, nosotros ahora podemos crear una instancia de la clase Morse llamado morse:

```
Morse morse(13);
```

Cuando esta línea se ejecuta (que en realidad sucede antes incluso de setup()), el constructor de la clase Morse será invocado y le pasara el argumento que se ha dado aquí (en este caso, sólo 13).
Tenga en cuenta que nuestra parte setup() del programa está vacía, porque la llamada a pinMode() se lleva a cabo en el interior de la librería (cuando la instancia se construye). Por último, para llamar a las funciones punto dot() y raya dash(), es necesario colocar el prefijo morse. – delante de la instancia que queremos usar. Podríamos tener varias instancias de la clase Morse, cada uno en su propio pin almacenados en la variable privada _pin de esa instancia.

Al llamar una función en un caso particular, especificaremos qué variables del ejemplo debe utilizarse durante esa llamada a una función. Es decir, si hemos escrito:

```
Morse morse(13);
Morse morse2(12);
```

entonces dentro de una llamada a morse2.dot(), _pin sería 12.
Si ha escrito el nuevo programa, probablemente se habrá dado cuenta de que ninguna de nuestras funciones de la librería fue reconocida por el entorno de Arduino destacando su color. Por desgracia, el software de Arduino no puede averiguar automáticamente lo que se ha definido en su librería (a pesar de que sería una característica interesante), por lo que tiene que darle un poco de ayuda. Para hacer esto, cree un archivo llamado keywords.txt en el directorio Morse. Debe tener este aspecto:

```
Morse	KEYWORD1
dash	KEYWORD2
dot	KEYWORD2
```

Cada línea tiene el nombre de la palabra clave, seguida de un tabulador (sin espacios), seguido por el tipo de palabra clave. Las clases deben ser KEYWORD1 y son de color naranja; las funciones deben ser KEYWORD2 y serán de color marrón. Tendrás que reiniciar el entorno Arduino para conseguir reconocer las nuevas palabras clave.
Es interesante que quienes utilicen la libreria Morse tengan algún ejemplo guardado y que aparezca en el IDE de Arduino cuando seleccionamos dentro de la carpeta ejemplos (Sketch). Para hacer esto, se crea una carpeta de ejemplos dentro de la carpeta que contiene la librería Morse. A continuación, movemos o copiamos el directorio que contiene el programa (lo llamaremos SOS) que hemos escrito anteriormente en el directorio de ejemplos. (Usted puede encontrar el ejemplo mediante el menú Sketch → Show Sketch Folder). Si reiniciamos el entorno de Arduino veremos un elemento Library_Morse dentro del menú File → Sketchbook → Examples que contiene su ejemplo. Es posible que desee añadir algunos comentarios que explicar mejor cómo utilizar la biblioteca.

Apéndice C
Señales analógicas de salida en Arduino (PWM)

En este apartado vamos a ver los fundamentos en los que se basa la generación de salidas analógicas en Arduino. El procedimiento para generar una señal analógica es el llamado PWM. Señal PWM (Pulse-width modulation) señal de modulación por ancho de pulso.
Donde:

- PW (Pulse Width) o ancho de pulso, representa al ancho (en tiempo) del pulso.
- length/period (periodo), o ciclo , es el tiempo total que dura la señal.

La frecuencia se define como la cantidad de pulsos (estado on/off)por segundo y su expresión matemática es la inversa del periodo, como muestra la siguiente ecuación.

$$Frequency = \frac{1}{period} \tag{C.1}$$

El periodo se mide en segundos, de este modo la unidad en la cual se mide la frecuencia (hertz) es la inversa a la unidad de tiempo (segundos).
Existe otro parámetro asociado o que define a la señal PWM, denominado "Duty cycle", Ciclo de Trabajo, el cual determina el porcentaje de tiempo que el pulso (o voltaje aplicado) está en estado activo (on) durante un ciclo. Por ejemplo, si una señal tiene un periodo de 10 ms y sus pulsos son de ancho (PW) 2ms, dicha señal tiene un ciclo de trabajo (duty cycle) de 20 % (20 % on y 80 % off). El siguiente gráfico muestra tres señales PWM con diferentes "duty cycles".

La señal PWM se utiliza como técnica para controlar circuitos analógicos. El periodo y el ciclo de trabajo (duty cycle) del tren de pulsos puede

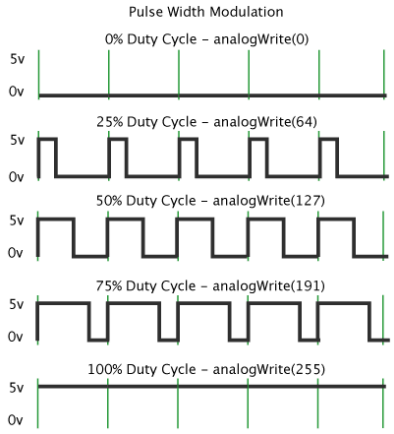

Figura C.1: Ancho de pulo

determinar la tensión entregada a dicho circuito. Si, por ejemplo, tenemos un voltaje de 5v y lo modulamos con un duty cycle del 10 %, obtenemos 0.5V de señal analógica de salida.
Las señales PWM son comúnmente usadas para el control de velocidad de motores DC (si decrementas el ciclo de trabajo sobre la señal de control del circuito de potencia que actúa sobre el motor el motor se mueve más lentamente), ajustar la intensidad de brillo de un LED, etc.
En Arduino, con ATmega168 o ATmega328, la señal de salida PWM (pines 3,5,6,9,10, y 11) es una señal de frecuencia 490 Hz aproximadamente y que sólo nos permite cambiar el "duty cycle.º el tiempo que el pulso está activo (on) o inactivo (off), utilizando la función analogWrite().
Otra forma de generar señales PWM es utilizando la capacidad del microprocesador. La señal de salida obtenida de un microprocesador es una señal digital de 0 voltios (LOW) y de 5 voltios (HIGH).
Con el siguiente código y con sólo realizar modificaciones en los intervalos de tiempo que el pin seleccionado tenga valor HIGH o LOW, a través de la función digitalWrite (), generamos la señal PWM.

```
/* senal PWM */

int digPin = 10; // pin digital 10

void setup()
{
    pinMode(digPin, OUTPUT);       // pin en modo salida

}

void loop() {
```

```
    digitalWrite(digPin, HIGH); // asigna el valor HIGH al pin
    delay(500);      // espera medio segundo
    digitalWrite(digPin, LOW); // asigna el valor LOW al pin
    delay(500);      // espera medio segundo
}
```

El programa pone el pin 10 a HIGH una vez por segundo durante medio segundo (ciclo de trabajo 50 %), la frecuencia que se genera en dicho pin es de 1 pulso por segundo o 1 Hz de frecuencia (periodo de 1 segundo). Cambiando la temporización del programa, podremos cambiar el ciclo de trabajo de la señal. Por ejemplo, si cambiamos las dos líneas con delay(500) por delay(250) y delay(750), modificamos el ciclo de trabajo a 25 %; ahora, el programa pone el pin 10 a HIGH una vez por segundo durante 1/4 de segundo y la frecuencia sigue siendo de 1 Hz.
Utilizando la función analogWrite(pin,value) podemos obtener la misma señal a una frecuencia de 490 Hz aproximadamente. Para una señal PWM con ciclo de trabajo 50 % hay que poner en el parámetro value, de la función analogWrite(pin,value), el valor de 127.

```
/* senal PWM en el pin 10 de ciclo de trabajo 50*/

int digPin = 10; // pin digital 10

void setup()
{
                // no se declara el modo del pin
                //como salida analogica
}

void loop() {
    analogWrite(digPin,127); // Senal PWM a 50% en el PIN 10
}
```

De forma que cambiando el valor del parámetro value en la función analogWrite(pin,value), podemos obtener distintos ciclos de trabajo:

value	Ciclo de trabajo
0	0 %
63	25 %
127	50 %
190	75 %
255	100 %

Cuadro C.1: Modificación de ciclos de trabajo

Apéndice D
Comunicando Arduino con otros sistemas

Hoy en día la manera más común de comunicación entre dispositivos electrónicos es la comunicación serial y Arduino no es la excepción. A través de este tipo de comunicación podremos enviar datos a y desde nuestro Arduino a otros microcontroladores o a un computador corriendo alguna plataforma de medios (Processing, PD, Flash, Director, VVVV, etc.). En otras palabras conectar el comportamiento del sonido o el video o un programa monitor a sensores o actuadores. Explicaré aquí brevemente los elementos básicos de esta técnica:

D.1. Funciones básicas

El mismo cable con el que programamos el Arduino desde un computador es un cable de comunicación serial. Para que su función se extienda a la comunicación durante el tiempo de ejecución, lo primero es abrir ese puerto serial en el programa que descargamos a Arduino. Para ello utilizamos la función:

```
Serial.begin(9600);
```

Ya que solo necesitamos correr esta orden una vez, normalmente iría en el bloque void setup(). El número que va entre paréntesis es la velocidad de transmisión y en comunicación serial este valor es muy importante ya que todos los dispositivos que van a comunicarse deben tener la misma velocidad para poder entenderse. 9600 es un valor estándar y es el que tienen por defecto Arduino al iniciar.
º Una vez abierto el puerto lo más seguro es que luego queramos enviar al

computador los datos que vamos a estar leyendo de uno o varios sensores. La función que envía un dato es:

```
Serial.print(data);
```

Una mirada en la referencia de Arduino permitirá constatar que las funciones print y println (lo mismo que la anterior pero con salto de renglón) tienen opcionalmente un modificador que puede ser de varios tipos:

- Serial.print(data, DEC); // decimal en ASCII
- Serial.print(data, HEX); // hexadecimal en ASCII
- Serial.print(data, OCT); // octal en ASCII
- Serial.print(data, BIN); // binario en ASCII
- Serial.print(data, BYTE); // un Byte

Como puede verse, prácticamente todos los modificadores, menos uno, envían mensajes en ASCII. Explicaré brevemente:

D.2. Series de pulsos

En el modo más sencillo y común de comunicación serial (asincrónica, 8 bits, más un bit de parada) siempre se está enviando un byte, es decir un tren de 8 pulsos de voltaje legible por la máquina como una serie de 8 bit (1 ó 0):

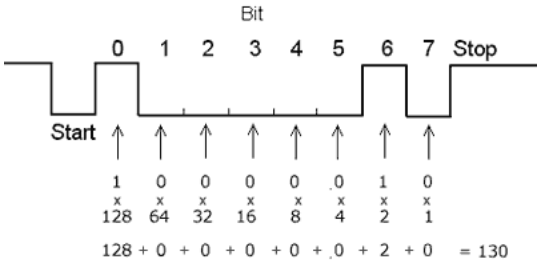

Figura D.1: Series de Pulsos

O sea que no importa cual modificador usemos siempre se están enviando bytes. La diferencia esta en lo que esos bytes van a representar y sólo hay dos opciones en el caso del Arduino: una serie de caracteres ASCII o un número.

dato	Modificador	Envío (pulsos)
65	DEC	(’6’ y ’5” ASCIIs 54–55)
65	HEX	(’4’ Y ’1’ ASCIIs 52–49)
65	OCT	(’1’, ’0’ y ’1’ ASCIIs 49–48–49)
65	BIN	(’0’,’1’,’0’,’0’,’0’,’0’,’0’y ’1’ ASCIIs 49–48–49–49–49–49–49–48)
65	BYTE	01000001

Cuadro D.1: Equivalencia a binario

Si Arduino lee en un sensor analógico un valor de 65, equivalente a la serie binaria 01000001 esta será enviada, según el modificador, como: No explicaremos conversiones entre los diferentes sistemas de representación numérica, ni la tabla ASCII, pero es evidente como el modificador BYTE permite el envío de información más económica (menos pulsos para la misma cantidad de información) lo que implica mayor velocidad en la comunicación. Y ya que esto es importante cuando se piensa en interacción en tiempo real es el modo que usaremos.

D.3. Un ejemplo sencillo

Enviar un sólo dato es realmente fácil. En el típico caso de un potenciómetro conectado al pin 2 de Arduino:

```
int potPin = 2;
int ledPin = 13;
int val = 0;

void setup()
{
    Serial.begin(9600);
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, HIGH); //activamos el pin para saber
        cuando arranco
}

void loop()
{
    val = analogRead(potPin);      // lee el valor del Pot
    Serial.println(val);
}
```

Si no utilizamos ningún modificador para el Serial.println es lo mismo que si utilizáramos el modificador DEC. Así que no estamos utilizando el modo más

eficiente pero si el más fácil de leer en el mismo Arduino. Al ejecutar este programa podremos inmediatamente abrir el monitor serial del software Arduino (último botón a la derecha) y aparecerá el dato leído en el potenciómetro tal como si usáramos el println en Processing. Envío a Processing (versión ultra simple) Para enviar este mismo dato a Processing si nos interesa utilizar el modo BYTE así que el programa en Arduino quedaría así:

```
int potPin = 2;
int ledPin = 13;
int val = 0;

void setup()
{
    Serial.begin(9600);
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, HIGH); // activamos el pin para saber
        cuando arranco
}

void loop()
{
    // lee el Pot y lo divide entre 4 para quedar entre 0-255
    val = analogRead(potPin)/4;
    Serial.print(val, BYTE);
}
```

En Processing tenemos que crear un código que lea este dato y haga algo con él:

```
import processing.serial.*;

Serial puerto; // Variable para el puerto serial
byte pot; // valor entrante
int PosX;

void setup()
{
    size(400, 256);
    println(Serial.list()); // lista los puertos seriales
        disponibles

    //abre el primero de esa lista con velocidad 9600
    port = new Serial(this, Serial.list()[0], 9600);
    fill(255,255,0);
    PosX = 0;
    pot = 0;
}
```

```
void draw()
{
   if(puerto.available() > 0)
   { // si hay algun dato disponible en el puerto
     pot = puerto.read(); // lo obtiene
     println(pot);
   }
   ellipse(PosX, pot, 3, 3); // y lo usa
   if (PosX < width)
   {
      PosX++;
   } else { fill(int(random(255)),int(random(255)),int(random
      (255)));
    PosX = 0;
   }
}
```

Si ya se animó a intentar usar más de un sensor notará que no es tan fácil como duplicar algunas líneas.

D.4. Comunicación vía puerto Serie:

La tarjeta Arduino puede establecer comunicación serie (recibir y enviar valores codificados en ASCII) con un dispositivo externo, a través de una conexión por un cable/puerto USB o cable/puerto serie RS-232.
Igual que para la descarga de los programas, sólo será necesario indicar el número de puerto de comunicaciones que estamos utilizando y la velocidad de transferencia en baudios.También hay que tener en cuenta las limitaciones de la transmisión en la comunicación serie, que sólo se realiza a través de valores con una longitud de 8-bits (1 Byte)(ver Serial.write() o Serial.read(c)), mientras que como ya se hemos indicado, el A/D (Convertidor) de Arduino tiene una resolución de 10-bits.(enlace)
Dentro del interfaz Arduino, disponemos de la opción 'Monitorización de Puerto Serie', que posibilita la visualización de datos procedentes de la tarjeta. Para definir la velocidad de transferencia de datos, hay que ir al menú 'Herramientas' (Tools) y seleccionar la etiqueta 'Velocidad de monitor Serie'(Tools). La velocidad seleccionada, debe coincidir con el valor que hemos determinado o definido en nuestro programa y a través del comando Serial.begin().Dicha velocidad es independiente de la velocidad definida para la descarga de los programas.
La opción de 'Monitorización de puerto serie' dentro del entorno Arduino,

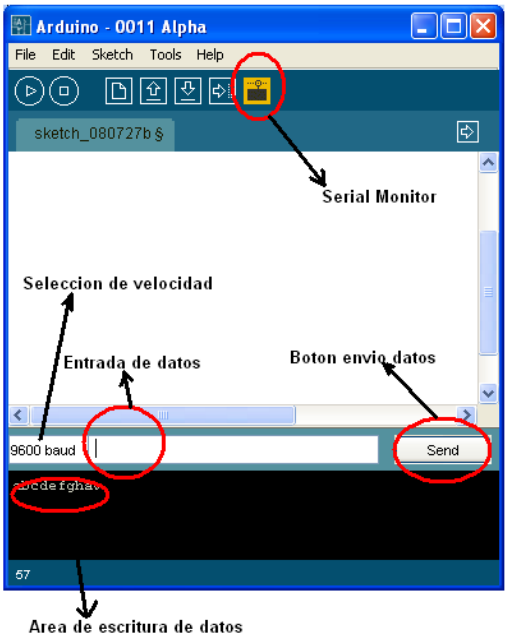

Figura D.2: Series de Pulsos

sólo admite datos procedentes de la tarjeta. Si queremos enviar datos a la tarjeta, tendremos que utilizar otros programas de monitorización de datos de puerto serie como HyperTerminal (para Windows) -Enlace o ZTerm (para Mac)-XXXX- Linux-Enlace, etc.
También se pueden utilizar otros programas para enviar y recibir valores ASCII o establecer una comunicación con Arduino: Processing (enlace), Pure Data (enlace), Director(enlace), la combinación o paquete serial proxy + Flash (enlace), MaxMSP (enlace), etc.
Nota: Hay que dejar tiempos de espera entre los envíos de datos para ambos sentidos, ya que se puede saturar o colapsar la transmisión.

D.5. Envío de datos desde Arduino(Arduino->PC) al PC por puerto de comunicación serie:

Ejercicio de volcado de medidas o valores obtenidos de un sensor analógico

Código

```
/* Lectura de una entrada analogica en el PC
El programa lee una entrada analogica, la divide por 4
para convertirla en un rango entre 0 y 255, y envia el valor al
   PC en diferentes formatos ASCCI.
A0/PC5: potenciometro conectado al pin analogico 1 y puerto de PC
   -5
Created by Tom Igoe 6 Oct. 2005
Updated
*/

int val; // variable para capturar el valor del sensor analogico

void setup()
{
   // define la velocidad de transferencia a 9600 bps (baudios)
   beginSerial(9600);
}

void loop()
{
   // captura la entrada analogica, la divide por 4 para hacer el
       rango de 0-255
   val = analogRead(0)/4;

   // texto de cabecera para separar cada lectura:
   printString('Valor Analogico =');

   // obtenemos un valor codificado en ASCII (1 Byte) en formato
       decimal :
   printInteger(val);
   printString('\t'); //Caracter espacio

   // obtenemos un valor codificado en ASCII (1 Byte) en formato
       hexadecimal :
   printHex(val);
   printString('\t');

   // obtenemos un valor codificado en ASCII (1 Byte) en formato
       binario
   printBinary(val);
```

```
    printString('\t');

    // obtenemos un valor codificado en ASCII (1 Byte)en formato
        octal:
    printOctal(val);
    printString('\n\r'); //caracter salto de linea y retorno de
        carro

    // espera 10ms para la proxima lectura
    delay(10);
}
```

Otra solución puede ser la de transformar los valores capturados en un rango entre 0 y 9 y en modo de codificación ASCII o en caracteres ASCII. De forma que dispongamos de un formato más sencillo o legible, sobre la información capturada.
El siguiente código incluye una función llamada **treatValue()** que realiza dicha transformación.

```
int val; // variable para capturar el valor del sensor analogico

void setup()
{
    // define la velocidad de transferencia a 9600 bps (baudios)
    beginSerial(9600);
}

int treatValue(int data)
{
    return (data * 9 / 1024) + 48; //formula de transformacion
}

void loop()
{
    val= analogRead(0); //captura del valor de sensor analogico
        (0-1023)
    serialWrite(treatValue(val)); //volcado al puerto serie de 8-
        bits
    serialWrite(10); //caracter de retorno de carro
    serialWrite(13); //caracter de salto de linea
    delay(10);
}

// Serial Output
// by BARRAGAN <http://people.interaction-ivrea.it/h.barragan>

int switchpin = 0; // interruptor conectado al pin 0
```

```
void setup()
{
   pinMode(switchpin, INPUT); // pin 0 como ENTRADA
   Serial.begin(9600);      // inicia el puerto serie a 600bps
}

void loop()
{
   if(digitalRead(switchpin) == HIGH) //si el interruptor esta en
       ON
   {
      Serial.print(1);      // envia 1 a Processing
   } else {
      Serial.print(0);      // en caso contrario envia 0 a
          Processing
   }
   delay(100);       // espera 100ms
}
```

Apéndice E

Envío de datos desde el PC (PC->Arduino) a Arduino por puerto de comunicación serie

En primer lugar, necesitamos instalar un programa como Hyperterminal en nuestro PC, en caso de que sea Windows.....

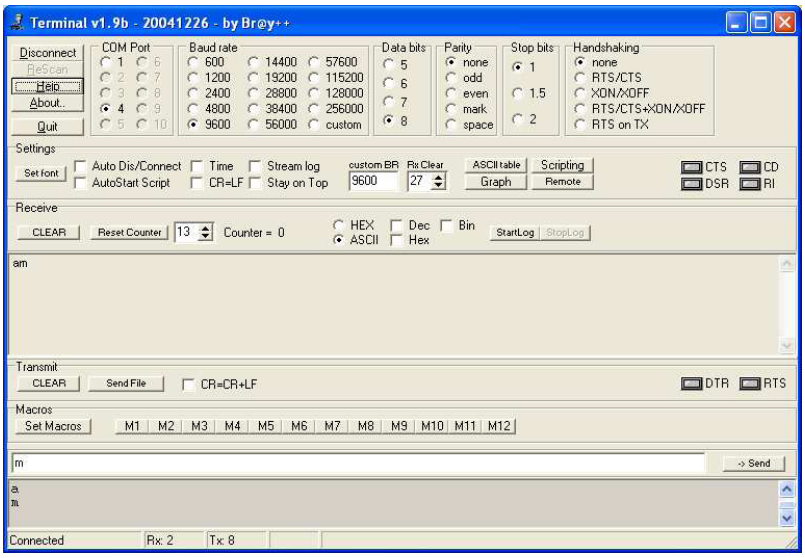

Figura E.1: Software terminal para realizar comunicaciones

Seleccionar el puerto que estamos utilizando con la tarjeta, la velocidad de transferencia y el formato de salida de los datos. Y finalmente conectar... Se puede realizar una comprobación con el siguiente programa mostrado a continuación. **Nota**: El programa de monitorización de datos está ocupando el puerto utilizado para la conexión a la tarjeta, por lo que si quieres realizar una nueva descarga del programa, tendrás que desconectarte previamente de

este último.

```
/*by BARRAGAN <http://people.interaction-ivrea.it/h.barragan>
*Demuestra como leer un dato del puerto serie. Si el dato
    recibido es una 'H', la luz se
*enciende ON, si es una 'L', la luz se apaga OFF. Los datos
    provienen del PC o de un
*programa como Processing..
*created 13 May 2004 revised 28 Aug 2005
*/

char val; // variable que recibe el dato del puerto serie
int ledpin = 13; // LED conectado al pin 13

void setup()
{
   pinMode(ledpin, OUTPUT); // pin 13 (LED)actua como SALIDA
   Serial.begin(9600);     // inicia la comunicacion con el puerto
        serie a 9600bps
}

void loop()
{
   if( Serial.available() )     // si hay dato e el puerto lo lee
   {
      val = Serial.read();     // lee y almacena el dato en 'val'
   }
   if( val == 'H' )     //su el dato recibido es 'H'
   {
      digitalWrite(ledpin, HIGH); //activa el LED
   } else {
      digitalWrite(ledpin, LOW); // en caso contrario lo
          desactiva
   }
   delay(100);      // espera 100ms para una nueva lectura
}
```

Para probar este programa bastará con iniciar el programa que actúe de "terminal de comunicación" Hyperterminal de Windowws o el programa mostrado anteriormente y podemos enviar los datos y comprobar como actúa.

E.1. Envío a petición (toma y dame)

Cuando se envía más de un dato del Arduino a otro sistema es necesario implementar reglas de comunicación adicionales para poder distinguir a que

dato corresponde cada uno de los paquetes de bytes recibidos. Una manera simple y eficiente de hacer esto es jugando al "toma y dame". Arduino no enviará los valores de los sensores hasta que Processing no le envíe también un valor por el puerto serial y Processing, a su vez, no enviara ese valor hasta no tener los datos que espera completos.
Este sería el código para Arduino usando tres potenciómetros en los últimos tres pines analógicos del ATmega: Codigo para cargar en la tarjeta Arduino

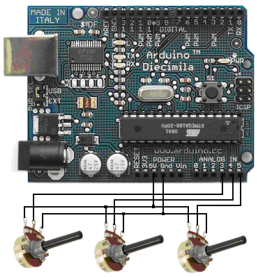

Figura E.2: Circuito de Arduino con potenciometros

desde el IDE Arduino

```
int pot1= 0;      // valores de los sensores analogicos
int pot2= 0;
int pot3= 0;
int inByte = 0;      // valor entrante de Processing

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

void loop()
{
   if (Serial.available() > 0) { // solo si algo ha llegado

      inByte = Serial.read();     // lo lee

      // hace la lectura de los sensores en pines 3,4y5 (analogos
          )

      pot1 = analogRead(3)/4; pot2 = analogRead(4)/4; pot3 =
          analogRead(5)/4;

      // y los envia

      Serial.print(pot1, BYTE); Serial.print(pot2, BYTE);
      Serial.print(pot3, BYTE);
```

```
    }
}
```

Una vez cargado este programa en la tarjeta Arduino está en disposición de enviar los datos de las lecturas de los potenciómetros cuando le sean demandados por el programa que los requiera. En nuestro ejemplo vamos a escribir un programa en el IDE Processing y será este el que se ocupe de leer los datos y con ellos modificar la posición de una bola que aparecerá en pantalla.
Será processing quién empezará el "toma y dame" y deberá reconocer cada dato. Este es el código:

Código para Processing

```
import processing.serial.*;

Serial puerto;
int[] datosEntrantes = new int[3]; // arreglo para recibir los
    tres datos
int cuantosDatos = 0;       // contador
int posX, posY, posZ;       // posicion de un objeto 3D
boolean hayDatos = false;       // control de verdad

void setup() { size(400, 400, P3D); noStroke();


println(Serial.list());// puertos serie disponibles
puerto = new Serial(this, Serial.list()[0], 9600); //
    Configuracion del puerto
puerto.write(65);     // Envia el primer dato para iniciar el toma
     y dame
}


void draw() {
background(0); lights(); fill(30,255,20);
translate(width/2 + posX, height/2 + posY, posZ);
sphere(40);

if (hayDatos == false) { //si no hay datos envia uno
puerto.write(65);
}
}
// esta funcion corre cada vez que llega un dato serial


void serialEvent(Serial puerto) {
```

```
if (hayDatos == false) {

hayDatos = true;      // de ahora en adelante el dato de envio se
    dara por el toma y
dame
}
// Lee el dato y lo anade al arreglo en su ultima casilla
datosEntrantes[cuantosDatos] = puerto.read();
cuantosDatos++;
if (cuantosDatos > 2 ) { // Si ya hay tres datos en el arreglo

posX = datosEntrantes[0]; posY = datosEntrantes[1]; posZ =
    datosEntrantes[2];

println("Valores de los potenciometros: " + posX + "," + posY + "
    ," + posZ);

puerto.write(65); // y envia para pedir mas

cuantosDatos = 0; // y todo empieza de nuevo
}

}
```

Aspecto del IDE Processing cuando está en funcionamiento el programa de captura de valores de los tres potenciómetros.

Índice de figuras