Instituto Politécnico

Universidad Nacional de Rosario Universidad Nacional de Rosario

Arduino

5º Año

Cód. 21501-16

Oscar Molinari
Luciano Zinni

Prácticas Profesionalizantes

Dpto. de Electrotecnia

Masterización: RECURSOS PEDAGOGICOS

ÍNDICE

1 INTRODUCCIÓN 3
1.1 ¿Qué es Arduino 3
1.2 Ejemplos de aplicación de Arduino 3
2 HARDWARE 5
2.1 Arduino Uno 5
2.1.1 Visión General 5
2.1.2 Resumen de características técnicas 6
2.1.3 Alimentación 6
2.1.4 Memoria 7
2.1.5 Entradas y salidas 7
2.1.6 Comunicación 8
2.1.7 Programación 8
2.1.8 Reseteo Automático (Software) 9
2.1.9 Protección de sobrecarga del USB 9
2.1.10 Características físicas 10
3 SOFTWARE Y PROGRAMACIÓN 11
3.1 Estructura de un programa 11
3.1.1 setup() 11
3.1.2 loop() 12
3.2 Punto y coma ; 12
3.3 Bloques de comentarios /*…*/ 12
3.4 Comentarios de línea // 13
3.5 Llaves {} 13
3.6 Variables 13
3.7 Tipos de datos 15
3.8 Aritmética 16
3.9 Operadores de comparación 17
3.10 Operadores lógicos 17
3.12 Control de flujo 18
3.12.1 Sentencia if 18
3.12.2 Sentencia if…else 18
3.12.3 Sentencia for 20
3.12.4 Sentencia while 21
3.12.5 Sentencia do…while 21
3.13 Funciones 22
3.14 E/S digital 23
3.14.1 pinMode(pin, modo) 23
3.14.2 digitalRead(pin) 23
3.14.3 digitalWrite(pin, value) 24
3.15 E/S analógicas 25
3.15.1 analogRead(pin) 25
3.15.2 analogWrite(pin, value) 25
3.16 Tiempo 26
3.16.1 delay(ms) 26

3.16.2 millis() 26
3.17 Matemáticas 26
3.17.1 min(x,y) 26
3.17.2 max(x,y) 27
3.18 Serie 27
3.18.1 Serial.begin(rate) 27
3.18.2 Serial.println(data) 27

1 INTRODUCCIÓN

1.1 ¿Qué es Arduino?

Arduino es una plataforma de prototipos electrónica de código abierto (open source) basada en hardware y software flexibles y fáciles de usar. Está pensado para artistas, diseñadores, como un hobby y para cualquiera interesado en crear objetos o entornos interactivos.

Arduino puede "sentir" el entorno mediante la recepción de señales desde una gran variedad de sensores y puede intervenir su alrededor mediante el control de luces, motores y otros artefactos. El microcontrolador de la placa se programa usando el "Arduino Programming Language" y el "Arduino Development Environment". Los proyectos de Arduino pueden ser autónomos o se pueden comunicar con software en ejecución en un ordenador.

Las placas se pueden ensamblar a mano[1] o encargarlas preensambladas. El software se puede descargar gratuitamente[2]. Los diseños de referencia del hardware (archivos CAD) están disponibles bajo licencia open source, por lo que se es libre de adaptarlas a las necesidades del interesado.

Existen otras plataformas y microcontroladores en sí que presentan similitudes con Arduino, pero este último tiene algunas ventajas. Por ejemplo, comparadas con las demás, las placas Arduino son relativamente baratas. El software de programación es multiplataforma, esto es, puede ejecutarse en distintos sistemas operativos, como Windows, GNU/Linux o Mac OS X. El mismo es muy simple y claro, fácil de usar para principiantes, pero a su vez, lo suficientemente flexible para que usuarios avanzados puedan sacar provecho de él.

Arduino está basado en microcontroladores de la marca Atmel. Los planos de los módulos están bajo licencia Creative Commons (es decir, son open source), lo que permite a diseñadores experimentados poder construir su propia versión, extendiéndolos y/o mejorándolos. Incluso usuarios relativamente inexpertos pueden construir la versión de la placa del módulo para entender cómo funciona y ahorrar dinero.

El lenguaje de programación básico es muy similar al lenguaje C++, por lo que una persona que sepa utilizarlo podrá crear un programa en Arduino sin ningún problema; bastará solamente que aprenda las funciones específicas del microcontrolador y su funcionamiento.

El software de Arduino, por ser de código abierto, puede ser expandido. La extensión del lenguaje puede hacerse mediante librerías en C++.

1.2 Ejemplos de aplicación de Arduino

Una de las ventajas de Arduino es su flexibilidad**.** Las placas pueden ser el cerebro de muchos sistemas, los cuales no necesariamente tienen que estas diseñados para tareas del mismo tipo. Por

ejemplo, Arduino es uno de los más elegidos por programadores y diseñadores de robots. Esto se debe a que el consumo de la placa por sí misma es relativamente poco, comparado con otros sistemas equivalentes, lo que permite alargar el tiempo de uso de circuitos del sistema que requieran mayor potencia debido a que éste puede aprovechar la carga de la batería mejor.

Otro gran uso que se le puede dar a Arduino es en aplicaciones de domótica. De nuevo, el bajo consumo de la placa y el fácil manejo de las entradas y salidas del mismo hacen que sea ideal a la hora de diseñar una casa verde, por ejemplo. Una casa verde es una casa que toma la energía de fuentes renovables, como energía eólica o solar. Estos tipos de energías en general son caros a la hora de la instalación, ya que para abastecer a una casa es necesario contar con varios paneles solares, o molinos enormes, por ejemplo, y la potencia que pueden entregar es la justa y necesaria. Por lo tanto, se necesita que los sistemas de control de la casa consuman por si mismos la menor energía posible para destinarla en su mayoría a los electrodomésticos u otros aparatos que sean mayores consumidores, de manera de no preocuparse sobre la potencia disponible.

Una de las desventajas que tiene Arduino es que no está destinado a ambientes industriales. No cuenta con las normas de seguridad que sí cuentan por ejemplo los PLC, y es fácil que falle en ese ámbito.

2 HARDWARE

Hay múltiples versiones de la placa Arduino, el curso se centrará únicamente en una sola: Arduino Uno.

2.1 Arduino Uno

2.1.1 Visión General

El Arduino Uno es una placa microcontroladora basada en el ATmega328. Tiene 14 pines de entrada/salida (E/S) digital (de los cuales 6 pueden ser usados como salidas PWM), 6 entradas analógicas, un oscilador de cuarzo a 16MHz, una conexión USB, un conector para alimentación, una cabecera ICSP, y un botón de reset. La placa contiene todo lo necesario para soportar el microcontrolador; con sólo conectarla a un ordenador a través del puerto USB o enchufarla con un adaptador AC/DC a la red eléctrica, o a una batería se puede comenzar a utilizar.

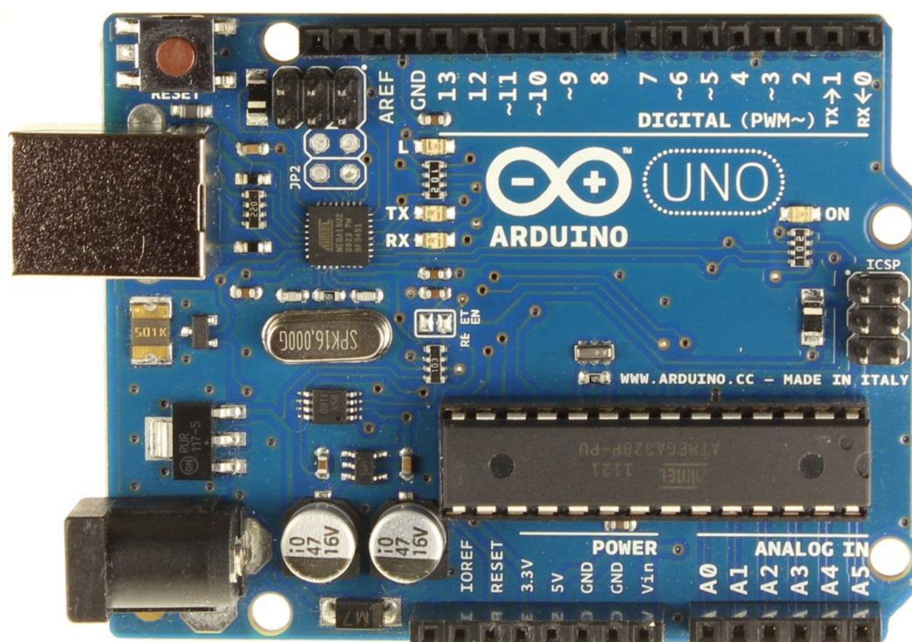

Fig 2.1. Imagen de la placa Arduino Uno.

2.1.2 Resumen de características técnicas

Característica	Descripción
Microcontrolador	ATmega328
Voltaje de operación	5V
Tensión de entrada (recomendada)	7-12V
Tensión de entrada (límite)	6-20V
Pines digitales de E/S	14 (de los cuales 6 proveen salidas PWM)
Pines de entrada analógicos	6
Corriente máxima DC por pin E/S	40 mA
Corriente máxima DC por pin 3.3V	50 mA
Memoria Flash	32 KB (de los cuales 0.5 KB es usado por el bootloader)
SRAM	2 KB
EEPROM	1 KB
Frecuencia de reloj	16 MHz

Tabla 2.1. Características técnicas de Arduino Uno.

2.1.3 Alimentación

El Arduino Uno puede ser alimentado a través de la conexión USB o con un suministro de energía externo. La fuente de energía es seleccionada automáticamente. La alimentación externa (no USB) puede provenir desde un adaptador AC/DC de pared o desde una batería. En el caso del adaptador, este puede conectarse mediante un enchufe de 2.1mm centro-positivo al conector de alimentación de la placa. Los cables de la batería pueden ser insertados en las cabeceras de los pines GND y Vin del conector POWER.

La placa puede operar con una alimentación externa de 6 a 20 voltios. Si la tensión suministrada es menor a 7V, sin embargo, el pin de 5V puede suministrar menos de cinco voltios y la placa podría ser inestable. Si se usa más de 12V, el regulador de voltaje podría sobrecalentarse y dañar la placa. El rango recomendado es de 7 a 12 voltios.

Los pines del conector de alimentación son los siguientes:

- **Vin**: La entrada de tensión a la placa Arduino cuando se está usando una fuente de alimentación externa (al contrario de los 5V de la conexión USB u otra fuente de alimentación regulada). Se puede suministrar tensión a través de este pin, o si ya se está suministrando tensión a través del conector jack, se puede acceder a él a través de este pin.
- **5V**: Este pin suministra 5V regulados del regulador de tensión de la placa. Esta puede ser alimentada desde el conector jack (7-12V), la conexión USB (5V), o del pin Vin de la placa (7-12V). Si se alimenta la placa a través del pin 5V o del 3.3V sin pasar por el regulador, se puede dañar a la misma. Esto último no es recomendable.
- **3V3**: Un suministro de 3.3V generado por el regulador de la placa. La corriente máxima que se le puede extraer es de 50mA.
- **GND**: Pines de tierra.
- **IOREF**: Este pin en la placa Arduino provee la referencia de voltaje con la que el microcontrolador opera. Una placa bien configurada puede leer el voltaje del pin IOREF y seleccionar la fuente de alimentación apropiada o activar el interpretador de voltaje en las salidas para trabajar con 5V o 3.3V.

2.1.4 Memoria

El ATmega328 tiene 32KB (de los cuales 0,5KB son usados por el bootloader). Además tiene 2KB de SRAM y 1KB EEPROM (que puede ser leída y escrita con la librería EEPROM).

2.1.5 Entradas y salidas

Cada uno de los 14 pines del Uno pueden ser usados como entrada o salida, usando las instrucciones *pinMode()*, *digitalWrite()*y *digitalRead()*. Ellos operan a 5 voltios. Cada pin puede proporcionar o recibir un máximo de 40mA y tiene una resistencia interna “pull-up” (desconectada por defecto) de 20-50 KOhms. Además, algunos pines tienen funciones especiales:

- Serial: 0 (RX) y 1 (TX). Usados para recibir (RX) y transmitir (TX) datos TTL en serie. Estos pines están conectados a los pines correspondientes del chip ATmega8U2 USB-a-TTL Serie.
- Interruptores externos: 2 y 3. Estos pines pueden ser configurados para disparar un interruptor en un valor bajo, un flanco creciente o decreciente o un cambio de valor.
- PWM: 3, 5, 6, 9, 10 y 11. Proporcionan salidas PWM de 8 bits con la función *analogWrite()*.
- SPI: 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). Estos pines soportan comunicación SPI usando la librería SPI.
- LED: 13. Hay un LED empotrado conectado al pin digital 13. Cuando el pin está en valor HIGH, el LED está encendido, cuando el pin está en valor LOW, está apagado.

El Uno tiene 6 entradas analógicas, nombradas desde A0 hasta A5, las cuales proporcionan 10 bits de resolución (por ejemplo, 1024 valores diferentes). Por defecto ellas miden desde tierra hasta 5 voltios, aunque es posible cambiar el valor mayor del rango utilizando el pin AREF y la función *analogReference()*. Adicionalmente, algunos pines tienen funcionalidades especiales:

- TWI: A4 o SDA pin y A5 o SCL pin. Soportan comunicación I^2C (o TWI) usando la librería Wire.

Hay algunos pines más en la placa:

- AREF. Voltaje de referencia para entradas analógicas. Se usa con la función *analogReference()*.
- Reset. Poniendo esta pin al estado LOW se resetea el microcontrolador. Típicamente se usa para añadir un botón de reset a dispositivos que bloquean al que se encuentra en la placa principal.

2.1.6 Comunicación

El Arduino Uno tiene un número de infraestructuras para comunicarse con un ordenador, otro Arduino u otros microcontroladores. El ATmega328 provee comunicación serie UART TTL (5V), la cual está disponible en los pines digitales 0 (RX) y 1 (TX). Un chip ATmega16U2 en la placa canaliza esta comunicación serie al USB y aparece como un puerto COM virtual en el software del ordenador. El firmware de este chip usa drivers estándar USB COM, por lo que no es necesario ningún driver externo. Sin embargo, en Windows suele fallar la instalación de ellos y por lo tanto se necesita realizar algunos pasos más. El software de Arduino incluye un monitor serial que permite enviar desde y a la placa Arduino simples datos de texto. Los LED's RX y TX en la placa titilarán cuando hay datos transfiriéndose vía el chip USB-a-serie y la conexión USB a la computadora (pero no para comunicación serie en los pines 0 y 1).

Una librería *SoftwareSerial* permite comunicación serie en cualquiera de los pines digitales del Uno.

El Atmega328 además soporta comunicación I^2C y SPI. El software de Arduino incluye una librería Wire para simplificar el uso del bus I^2C.

2.1.7 Programación

El Arduino Uno puede ser programado con el software de Arduino. El ATmega328 en la placa del Uno viene con el bootloader pregrabado, lo que permite subirle nuevo código sin la utilización de un programador de hardware externo. Se comunica utilizando el protocolo original STK500.

Además se puede saltar el bootloader y programar el microcontrolador a través de la cabecera ICSP (por sus siglas en inglés de In-Circuit-Serial-Programming). El código fuente del firmware del Atmega16U2 (o el 8U2 en la versión primera y segunda del Uno) está disponible.

2.1.8 Reseteo Automático (Software)

En lugar de requerir una pulsación física del botón de reset antes de una subida, el Arduino Uno está diseñado de una manera que permite ser reseteado por software corriendo en una computadora conectada. Una línea del control de flujo de hardware (DTR) del ATmega8U2/16U2 está conectada con la línea de reseteo del ATmega328 a través un capacitor de 100nF. Cuando esta línea toma el valor LOW, la línea de reseteo se mantiene el tiempo suficiente para resetear el chip. El software Arduino usa esta capacidad para permitir cargar código simplemente presionando el botón de subida en el entorno Arduino. Esto significa que el bootloader puede tener un tiempo de espera más corto, así como la puesta a LOW del DTR puede estar bien coordinada con el comienzo de la carga.

Esta configuración tiene otras implicancias. Cuando el Uno está conectado a una computadora que corre Mac OS X o Linux, se resetea cada vez que una conexión a él es hecha mediante software (vía USB). Durante el siguiente medio segundo aproximadamente, el bootloader se ejecutará en el Arduino Uno. Mientras está programado para ignorar los datos "malformados" (por ejemplo, cualquiera excepto una carga de código nuevo), interceptará los primeros bytes de datos enviados a la placa luego de que la conexión se abre. Si una rutina en la placa que se está corriendo recibe una configuración una vez u otros datos cuando empieza, hay que asegurarse que el software con el que se comunica espere un segundo luego de abrir la conexión y antes de enviar datos.

El Uno contiene una pista de soldadura que puede ser cortada para desactivar el auto-reset. Los caminos de cada lado de la pista pueden ser soldados en conjunto para restablecerlo. Está etiquetado como "RESET-EN". También es posible deshabilitar el auto-reset conectando una resistencia de 110 ohms desde los 5V a la línea de reseteo.

2.1.9 Protección de sobrecarga del USB

El Arduino Uno tiene un fusible reseteable que protege el puerto USB de la computadora de cortes y sobrecargas. Aunque la mayoría de los ordenadores proveen su propia protección interna, el fusible proporciona una capa de protección extra. Si más de 500mA se aplican al puerto USB, el fusible romperá automáticamente la conexión hasta que el corte o sobrecarga sean eliminados.

2.1.10 Características físicas

El máximo de longitud y anchura del PCB del Uno es de 2.7 y 2.1 pulgadas (6.86 y 5,33 centímetros) respectivamente, con el conector USB y el conector de alimentación que se extienden más allá de las dimensiones especificadas. Cuatro agujeros de tornillo le permiten a la placa atornillarse a una superficie o caja.

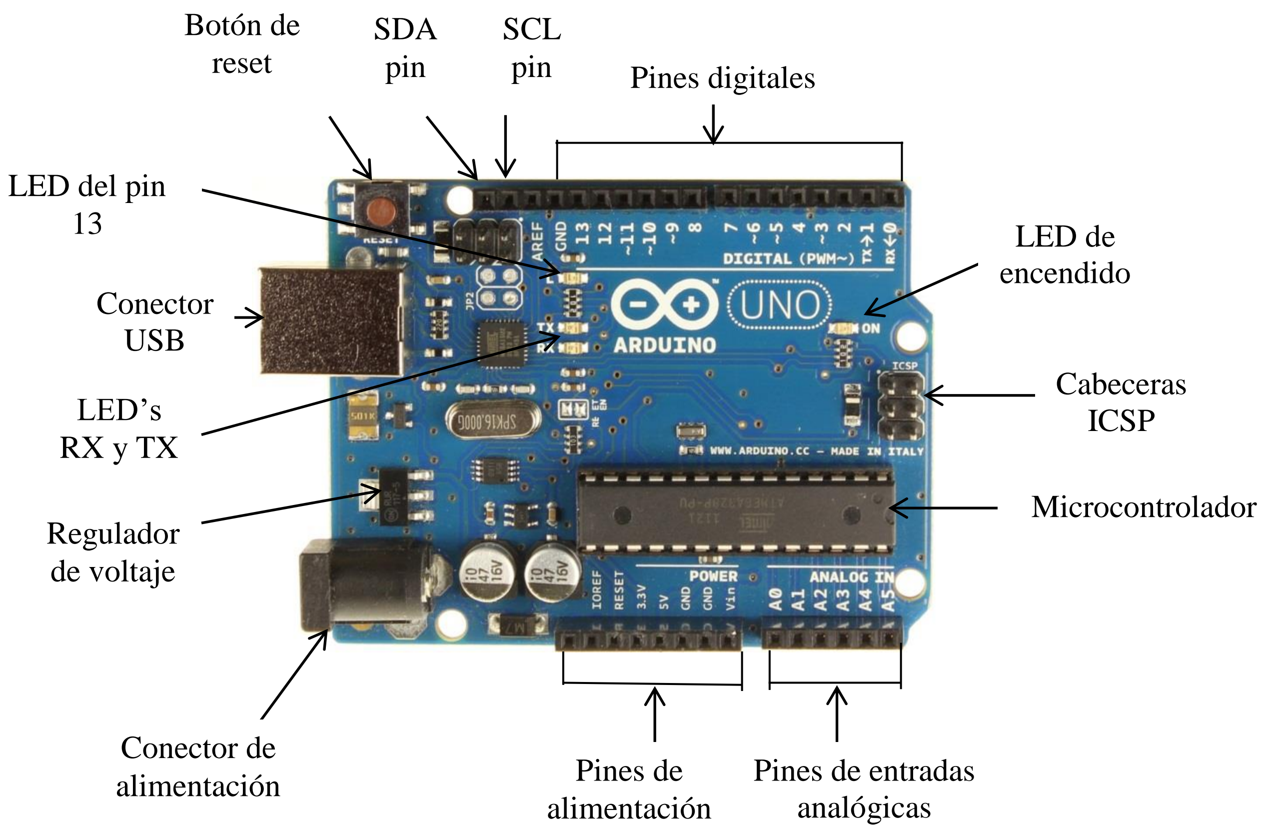

Fig 2.2. Imagen ilustrativa de los componentes de la placa Arduino Uno.

3 SOFTWARE Y PROGRAMACIÓN

Para esta sección se presupone ya instalado el entorno de programación Arduino, con los drivers correspondientes a la placa Uno. Si no se ha realizado este procedimiento, la guía para el mismo se encuentra en la página oficial de Arduino.

3.1 Estructura de un programa

La estructura básica del lenguaje de programación Arduino es bastante simple y se organiza en al menos dos partes o funciones que encierran los bloques de declaraciones:

```
void setup ()
{
    declaraciones;
}
void loop ()
{
    declaraciones;
}
```

Ambos funciones (*setup()* y *loop()*) son requeridas para que el programa funcione.

3.1.1 setup()

La función *setup()* es ejecutada una vez en la iniciación del programa y es usada para asignar la función a los pines digitales con la función *pinMode()* o inicializar las comunicaciones en serie, por ejemplo.

Ejemplo 3.1:

```
void setup ()
{
    pinMode(3,OUTPUT); //declara al pin digital 3 como salida
}
```

En general, dentro de esta función se encuentran las instrucciones que realizan la configuración inicial del programa y de la placa.

3.1.2 loop()

La función *loop()* se ejecuta luego de la anterior descrita, siendo el núcleo de todos los programas Arduino y haciendo la mayor parte del trabajo. Contiene el código que continuamente se va ejecutando, leyendo el estado de entradas, activando o desactivando salidas, tomando decisiones, etc.

Ejemplo 3.2:

```
void loop()
{
    digitalWrite(3,HIGH);     //activa el pin 3 (previamente declarado como salida)
    delay(1000);              //espera un segundo
    digitalWrite(3,LOW);      //desactiva el pin 3
    delay(1000);              //espera un segundo
}
```

Una vez que se ejecutó la función *setup()*, el programa entra en la función *loop()* ejecutando cada una de sus declaraciones. Al llegar a la última, salta a la primer declaración dentro de *loop()* y las realiza de nuevo, hasta llegar a la última; luego vuelve a la primera y así sucesivamente. De esta manera, las declaraciones dentro de *loop()* se ejecutan una y otra vez mientras el programa esté corriendo.

3.2 Punto y coma ;

Un punto y coma debe usarse al final de cada declaración y separa los elementos del programa. También se usa para separar elementos en un bucle *for*. En el caso de olvidar un punto y coma al final de una declaración producirá un error de compilación.

3.3 Bloques de comentarios /*…*/

Los bloques de comentarios o comentarios multilínea, son áreas de textos ignoradas por el programa y se usan para grandes descripciones de código o comentarios que ayudan a otras personas a entender partes del programa. Empiezan con /* y terminan con */, y pueden abarcar múltiples líneas.

```
/*
este es un bloque de comentario encerrado
no hay que olvidar cerrar el comentario
tienen que estar balanceados!
```

*/

Como los comentarios son ignorados por el programa y no ocupan espacio en memoria deben usarse generosamente y también pueden usarse para "comentar" bloques de códigos con propósitos de depuración.

3.4 Comentarios de línea //

Los comentarios de una línea empiezan con // y terminan con la siguiente línea de código. Como el bloque de comentarios, son ignorados por el programa y no toman espacio en memoria.

//este es un comentario de una línea

Comentarios de una línea se usan a menudo después de declaraciones válidas para proporcionar más información sobre qué lleva la declaración o proporcionar un recordatorio en el futuro.

3.5 Llaves {}

Las llaves definen el comienzo y final de bloques de función y bloques de declaraciones como *loop()* y la sentencia *for*. Las llaves deben estar balanceadas (a una llave de apertura { debe seguirle luego de las declaraciones correspondientes una de cierre }). Las llaves no balanceadas provocan errores de compilación.

```
void loop()

{

        declaraciones;

}
```

Cuando dentro de un bloque la declaración es una sola, las llaves pueden omitirse.

El entorno Arduino incluye una práctica característica para chequear el balance de llaves. Sólo es necesario seleccionar una llave y su compañera lógica aparecerá resaltada.

3.6 Variables

Una variable simboliza un objeto que tiene asociado una dirección o localización en memoria y un valor o "contenido" que puede cambiar durante la ejecución de un programa. Posee los siguientes atributos: un nombre que la designa y un tipo que describe su uso y el tipo de dato a contener.

Al tener asociado un valor, es necesario un procedimiento para asignarles a las variables los valores, esto es, se necesita una operación de asignación. El operador de asignación en el lenguaje Arduino es el símbolo "=" ("igual") y la declaración correspondiente es:

variable = valor;

Ejemplo 3.3:

Asignar a la variable con nombre AGRAV el valor 9.8. La sentencia para este ejemplo sería entonces:

AGRAV = 9.8;

Notar que la asignación es de derecha a izquierda, es decir, el valor 9.8 es asignado a AGRAV. Luego, en cualquier parte del programa, si quiero usar el valor 9.8, en lugar de escribirlo de esa manera se puede usar el nombre de la variable AGRAV.

Una variable puede ser declarada al comienzo del programa (antes del *void setup()*), localmente dentro de funciones, y algunas veces en un bloque de declaración, como por ejemplo, bucles *for*. Donde la variable es declarada determina el ámbito de la misma, o la habilidad de ciertas partes de un programa para hacer uso de ella.

Una variable global es una que puede ser vista y usada por cualquier función y declaración en un programa. Esta variable se declara al comienzo del programa, antes de la función *setup()*.

Una variable local es una que se define dentro de una función en la cual es declarada. Además, es posible tener dos o más variables del mismo nombre en diferentes partes del programa que contienen valores distintos.

```
int value; //"value" es visible por cualquier función

void setup()

{

        //no se necesita setup

}

void loop()

{

        for(int i=0; i<20;) //"i" es sólo visible dentro del bucle for

        {

                i++;

        }


        float f; //"f" es sólo visible dentro de loop()


}
```

3.7 Tipos de datos

En la sección anterior se habló de que las variables tienen un tipo de dato asociado. El tipo de una variable describe sus posibilidades de uso y la cantidad de memoria que el programa va a utilizar para almacenarla.

byte: Almacena un valor numérico de 8 bits sin puntos decimales. Tienen un rango de 0 a 255.

byte someVariable = 180; //declara "someVariable" como tipo byte y le asigna el valor 180

int: las variables de tipo int o enteros almacenan datos numéricos sin puntos decimales de 16 bits. Su rango va desde -32768 a 32767.

int someVariable=1500; //declara "someVariable" como tipo int y le asigna el valor 1500

long: Tipo de datos de tamaño extendido para enteros largos, sin puntos decimales, almacenados en un valor de 32 bits con un rango de -2146483648 a 2146483647.

long someVariable = 90000 //la declara como tipo long y le asigna 90000

float: Un tipo de datos para números en punto flotantes, es decir, números que tiene parte decimal. Los números en punto flotante tienen mayor resolución que los enteros y se almacenan como valor de 32 bits con un rango de $-3.4028235x10^{38}$ a $3.4028235x10^{38}$.

En el ejemplo de la sección anterior, la variable AGRAV debe ser de tipo float porque si se le quiere asignar el valor 9.8, tiene que poder contener datos de tipo numéricos con punto decimal.

float AGRAV = 9.8 //declara AGRAV como tipo float y le asigna el valor 9.8

boolean: Los tipos de datos booleanos tienen sólo dos posibles estados: verdadero (TRUE) o falso (FALSE).

boolean "someVariable" = TRUE; //declara como tipo boolean y le asigna TRUE

void: son datos que no necesitan un tipo en específico porque no serán almacenadas en memoria.

array: Un array es una colección de valores que son accedidos con un índice numérico. Cualquier valor en el array debe llamarse escribiendo el nombre del mismo y el índice del valor. Al primer valor del array se accede con el índice 0. Un array necesita ser declarado y opcionalmente asignarle valores antes de que puedan ser usados.

int myArray[]= {valor1, valor2, valor3, …};

Asimismo, es posible declarar un array especificando el tipo del mismo y su tamaño, y luego asignarle valores a una posición del índice.

int myArray[5]; //declara un array de tipo entero (int) con 6 posiciones

myArray[3]=10; //asigna a la cuarta posición del índice el valor 10

Para asignarle a otra variable el valor de una posición de un array, se necesita especificar la posición del índice y el nombre del array.

x = myArray[3]; //x ahora es igual a 10

Nota 1: Existen otros tipos de datos que aquí no se especifican ya que son usados con menos regularidad en los programas de Arduino, como por ejemplo, el tipo *char*.

Nota 2: Los tipos de datos int, long y float abarcan números negativos también. Si se quisiese trabajar con enteros positivos únicamente, se debe anteponer la palabra *unsigned* en la declaración, dejando la memoria disponible para números positivos nomás. En el caso del tipo int sin signo, por ejemplo, el rango va desde los 0 a 65535.

unsigned long someVariable = 90000;

3.8 Aritmética

Los operadores aritméticos incluyen suma (+), resta (-), multiplicación (*) y división (/). Retornan la suma, diferencia, producto o cociente, respectivamente, de dos operandos.

y = y+3;

x = x-7;

i = i*y;

r = r/23;

La operación es llevada a cabo usando el tipo de datos de los operandos, así, 9/4 devuelve 2 en lugar de 2.25, ya que 9 y 4 son de tipo enteros y no de tipo float. Si los operandos son de tipos diferentes, el tipo mayor es usado para el cálculo.

Cuando se realiza una operación aritmética y se quiere almacenar el resultado de ella en una variable que intervino en la misma operación, es posible acortar la escritura. Así, si se escribe y = y+3;, se puede acortar escribiendo y+=3. Esto indica que se le sume 3 a la variable y luego se asigne el resultado en ella.

x -= 7; // es lo mismo que escribir x = x-7;

i *= y; //es lo mismo que escribir i = i*y;

r /= 23; //es lo mismo que escribir r = r/23;

Además, si sólo se quiere sumar o restar uno al valor que contiene una variable, es posible acortar la escritura también.

y++; //equivale a escribir y = y+1; o y += 1;

b--; //equivale a escribir b = b-1; o b -= 1;

3.9 Operadores de comparación

Las comparaciones de una variable o constante con otra se usan a menudo en declaraciones donde se evalúa una condición entre ellas y se decide si es verdadera o falsa.

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

3.10 Operadores lógicos

Los operadores lógicos son normalmente una forma de comparar dos expresiones y devuelven TRUE o FALSE dependiendo del operador. Hay tres operadores lógicos: AND, OR y NOT.

AND:

x > 0 && x < 5 //verdadero sólo si las dos expresiones son ciertas

OR:

x > 0 || y > 0 //verdadero si al menos una expresión es cierta

NOT:

!(x > 0) //verdadero sólo si la expresión es falsa (niega la condición)

3.11 Constantes

El lenguaje Arduino tiene unos cuantos valores predefinidos que se llaman constantes. Se usan para hacer los programas más legibles. Las constantes se clasifican en grupos.

TRUE/FALSE:

Estas son constantes booleanas que definen niveles lógicos. FALSE se define como 0 y TRUE como 1 (o algún valor distinto de 0).

HIGH/LOW:

Estas constantes definen niveles lógicos de pines como HIGH y LOW y se usan cuando se leen o se escriben los pines digitales. HIGH está definido como el nivel 1 lógico, ON o 5V, mientras que LOW es el nivel lógico 0, OFF o 0V.

digitalWrite(13, HIGH); //enciende (pone a 1 o a 5V) el pin 13.

INPUT/OUTPUT:

Constantes usadas en la función *pinMode()* para definir el modo de un pin digital como entrada (INPUT) o salida (OUTPUT).

```
pinMode(13, OUTPUT); //declara como salida el pin 13
```

3.12 Control de flujo

3.12.1 Sentencia if

Las sentencias *if* comprueban si cierta condición es verdadera o falsa. En caso de ser cierta, se ejecutan todas las sentencias dentro de las llaves, de lo contrario, si es falsa, el programa las ignora y salta a la llave que cierra, continuando con el mismo.

En la condición se pueden comparar dos variables entre sí o una variable con un valor. Además, se pueden evaluar varias condiciones a la vez con los operadores lógicos AND, OR y NOT.

```
if(condición)
{
        declaraciones;
}
```

Ejemplo 3.4:

Evaluar si la variable entera de nombre "num" está entre los números 10 y 100 y prender el LED del pin 13 si esto resulta verdadero.

```
if (num>10 && num<100) //si num es mayor a 10 y menor a 100 la condición es verdadera
        digitalWrite(13, HIGH);
```

Nota 1: Recordar que si hay una sola declaración dentro del *if* no es necesario escribir las llaves.

Nota 2: Es muy común en los que se inician en el mundo de la programación de estos lenguajes confundir el operador de asignación "=" y el de comparación "==". Dentro del *if*, como se comparan elementos, se debe usar "==". Se recomienda prestar mucha atención a detalles como estos.

3.12.2 Sentencia if…else

If…else permite ejecutar otras sentencias en caso de que la condición sea negativa.

```
if(condición)
{
```

```
        declaraciones1;
}
else
{
        declaraciones2;
}
```

Si la comparación resulta verdadera, ejecuta las declaraciones1, y si resulta falsa, ejecuta las declaraciones2.

Else puede preceder a otra comparación *if*, por lo que múltiples y mutuas comprobaciones exclusivas pueden ejecutarse al mismo tiempo.

```
if(condición1)
{
        declaraciones1;
}
else if(condición2)
{
        declaraciones2;
}
else
{
        declaraciones3;
}
```

Ejemplo 3.5:

Si la variable booleana “hola” tiene el estado verdadero (TRUE), prender el LED del pin 13, de lo contrario, apagarlo.

```
if (hola)
        digitalWrite(13, HIGH);
else
```

digitalWrite(13, LOW);

Nota: Observar que en el ejemplo, no se comparó la variable "hola" con ninguna otra o con ningún valor. Esto se debe a que la misma variable contiene la información TRUE o FALSE, por lo que al colocarla dentro de una comparación, ésta ya contiene información de si es verdadera o no. Obviamente, el estado de "hola" se generó a partir de una comparación previa en el programa o porque el usuario le estableció su valor.

3.12.3 Sentencia for

La sentencia *for* se usa para repetir un bloque de declaraciones encerradas en llaves un número específico de veces. Un contador de incremento se usa a menudo para incrementar y terminar el bucle. Hay tres partes separadas por punto y coma (;), en la cabecera del bucle.

```
for (inicialización; condición; expresión)
{
        declaraciones;
}
```

La inicialización de una variable local, o contador de incremento, sucede primero y sólo una vez al iniciarse el primer bucle. Cada vez que pasa el bucle, la condición es comprobada. Si la condición devuelve TRUE, las declaraciones y expresiones dentro de las llaves se ejecutan y la condición se comprueba de nuevo. Cuando la condición se vuelve falsa (devuelve FALSE) el bucle termina. Además, antes de evaluar la condición nuevamente, ni bien se terminan de ejecutar todas las declaraciones del *for*, se ejecuta la expresión de la cabecera.

Paso por paso, la sentencia *for* funciona así:

1. "Inicialización" es ejecutada. Generalmente, aquí se declara una variable usada como contador, y se establece su valor inicial. Esta es ejecutada una sola vez, al principio del bucle.
2. "Condición" es evaluada. Si es verdadera, el bucle continúa; de lo contrario, el bucle termina y las declaraciones entre llaves son salteadas, yendo directo al paso 5.
3. Las declaraciones son ejecutadas. Como es usual, pueden ser una o varias declaraciones. Si son más de una, estas tienen que ir entre llaves.
4. "Expresión" es ejecutada, y el bucle vuelve al paso 2.
5. El bucle termina: la ejecución del programa continua con las declaraciones después de la sentencia *for*.

Ejercicio 3.1:

Se le propone al lector para evaluar la comprensión que trate de descubrir que es lo que hace el siguiente bloque de un programa de ejemplo.

```
for (int i=0; i<20, i++)
```

```
{
        digitalWrite(13, HIGH);
        delay(250);
        digitalWrite(13, LOW);
        delay(250);
}
```

3.12.4 Sentencia while

El bucle *while* se repetirá continuamente hasta que la condición que se evalúa devuelva FALSE. Primero se evalúa la condición, si es verdadera se ejecutan las declaraciones, luego se evalúa de nuevo la condición y así sucesivamente. Cuando la condición resulte falsa, el bucle termina.

```
while (condición)
{
        declaraciones;
}
```

Ejercicio 3.2:

Se le propone al lector que escriba por sí mismo un bloque de programa con la sentencia *while* que prenda y apague el LED del pin 13 diez veces.

3.12.5 Sentencia do…while

El bucle *do…while* trabaja de la misma forma que el bucle *while*, con la excepción de que la condición es testeada al final del bucle, por lo que el *do…while* siempre se ejecutará al menos una vez.

```
do
{
        declaraciones;
}while(condición)
```

Nota: Tanto para el bucle *while* y *do…while*, debe existir alguna declaración dentro de las llaves que en cierta pasada de la ejecución del bucle cambie la condición evaluada a falsa y termine. De no ocurrir esto, el programa nunca saldrá del bucle y las declaraciones dentro de él se ejecutarán indefinidamente. Para el bucle *for* ocurre lo mismo, sólo que puede pasar que la declaración que cambie el resultado de la condición se encuentre en la expresión de la cabecera y no dentro de las llaves.

3.13 Funciones

Una función es un bloque de código que tiene un nombre, un tipo (así como las variables) y un grupo de declaraciones que se ejecutan cuando se llama a la función. Se puede hacer uso de funciones integradas como *loop()* y *setup()* o escribir nuevas.

Las funciones se escriben para ejecutar tareas repetitivas y reducir el desorden en un programa. En primer lugar se declara el tipo de la función, que será el valor retornado por la misma (int, void, float, etc.). A continuación del tipo, se declara el nombre de la función y, entre paréntesis, los parámetros que pasan a ella.

La estructura de una función es entonces la siguiente:

```
tipo nombreFunción (parámetros)
{
        declaraciones;
}
```

Ejemplo 3.6:

La siguiente función tiene la tarea de devolver el resultado de elevar al número *a* a la potencia *b*:

```
int power (int a, int b)
{
        int n;
        for (n=1; b>0; b--)
                n*=a;
        return n;
}
```

A la función "power" se le pasan los parámetros *a* y *b*, estos son números enteros. Luego, se declara una variable entera temporal *n*, y a continuación se ejecuta el bucle *for*. Cuando se termina el bucle, se ejecuta la función *return*, que devuelve el valor final de *n*.

Nota: Para entender mejor este ejemplo, seguir los pasos que realiza el programa en papel, por ejemplo, tratando de calcular 2 elevado a la 5ta potencia (2^5).

Ejercicio 3.3:

Escribir una función llamada "factorial" que calcule el factorial de un número.

3.14 E/S digital

3.14.1 pinMode(pin, modo)

Se utiliza para configurar un pin específico para que se comporte como entrada (INPUT) o salida (OUTPUT).

pinMode(numPin, OUTPUT); //ajusta el pin "numPin" como salida

El parámetro numPin indica a que pin se está configurando. Se puede especificar el número directamente (2, 10, 13) o se puede poner el nombre de una variable de tipo entera que contenga el número del pin a configurar.

El parámetro modo indica si el pin a configurar actuará como salida o entrada. Si se desea que actúe como entrada, se debe escribir la constante INPUT; si se desea que actúe como salida, la constante OUTPUT.

Es usado generalmente en la función *setup()*, para realizar la configuración de todos los pines a usar. Sin embargo, es posible usarla dentro de *loop()*, pero en este caso hay que tener extremo cuidado al cambiar de modo un pin, ya que puede dañarse si se olvida cambiar también las sentencias que lo controlan.

Los pines digitales de Arduino están ajustados a INPUT por defecto, por lo que no necesitan ser declarados explícitamente como entradas con *pinMode()*. Los pines configurados como INPUT se dice que están en un estado de alta impedancia.

Hay también convenientes resistencias pull-up integradas en el chip ATmega que pueden ser accedidas por software:

pinMode(pin, INPUT); //ajusta "pin" como entrada

digitalWrite(pin, HIGH); //activa la resistencia pull-up

Las resistencias pull-up se usan generalmente para conectar entradas como interruptores.

Los pines configurados como OUTPUT se dice que están en un estado de baja impedancia y pueden proporcionar un máximo de 40mA a otros dispositivos o circuitos.

Nota: Cortocircuitos en los pines de Arduino o corriente excesiva pueden dañar o destruir el pin de salida o el chip ATmega. A menudo es buena idea conectar un pin OUTPUT a un dispositivo externo en serie con una resistencia de 470 Ohms o 1 KOhm.

3.14.2 digitalRead(pin)

Lee el valor desde un pin digital especificado y devuelve HIGH o LOW, según el estado del mismo. El pin puede ser especificado como una variable o una constante (0 a 13).

value = digitalRead(pinVariable); //asigna a "value" el valor del pin de entrada

3.14.3 digitalWrite(pin, value)

Le asigna el nivel lógico HIGH o LOW (es decir, activa o desactiva) a un pin digital. El pin puede ser especificado como una variable o constante (0 a 13).

digitalWrite(7, HIGH); //pone en alto el pin 7

Ejemplo 3.7:

Encender un LED al presionar un botón.

```
int led=13;     /*asigna el número 13 a la variable llamada led para futuras referencias en el
                programa*/

int pin=7;      //esté será el pin que esté conectado al botón

int value=0     //variable para almacenar el valor leído

void setup()

{

                pinMode(led, OUTPUT);        //ajusta el pin 13 como salida

                pinMode(pin, INPUT);         //ajusta el pin 7 como entrada

}

void loop()

{

                value = digitalRead(pin);    //lee el estado del pin de entrada 7 y lo asigna a
                                             //value

                digitalWrite(led, value);    //ajusta el estado del botón al pin "led" (13)

}
```

Nota 1: En el ejemplo se ha usado una variable tipo int a la que se le asignó el valor HIGH, que no es numérico. Se pudo hacer esto debido a que HIGH también puede asociarse a un valor 1 (o distinto de 0), así como también el valor TRUE. La constante LOW se puede asociar al valor 0 o, también, a FALSE.

Nota 2: Este programa necesita además del software, la conexión de un botón físico externo al pin 7 de la placa. Observar que no es necesario conectar un LED externo ya que la placa tiene empotrado uno que se conecta al pin 13. Como en el programa se usa el pin 13 como salida, el LED de la placa se encenderá cuando lo salida se encuentre en HIGH, y se apagará cuando la salida esté en LOW. Opcionalmente se puede conectar un LED externo con su resistencia correspondiente al pin 13.

3.15 E/S analógicas

3.15.1 analogRead(pin)

Lee el valor desde un pin analógico especificado con una resolución de 10 bits. Esta función sólo trabaja en los pines analógicos (0 a 5). Los valores enteros devueltos están en el rango de 0 a 1023.

value = analogRead(pin);   //asigna el valor de la entrada analógica "pin" a "value"

Nota: Los pines analógicos, al contrario de los digitales, no necesitan ser declarados al principio como INPUT o OUTPUT, siempre se usan como entradas.

3.15.2 analogWrite(pin, value)

Escribe un valor pseudo-analógico usando modulación por ancho de pulso (PWM, por sus siglas en inglés) a un pin de salida con capacidad de uso de PWM. En la placa Arduino Uno son los pines digitales 3, 5, 6, 9, 10 y 11. El valor puede ser especificado como una variable o constante con un valor de 0 a 255.

analogWrite(numPin, value); //escribe el valor de "value" en el pin "numPin"

Valor	Nivel de salida
0	0V (*t*)
64	0V (*3/4 de t*) y 5V (*1/4 de t*)
128	0V (*1/2 de t*) y 5V (*1/2 de t*)
192	0V (*1/4 de t*) y 5V (*3/4 de t*)
255	5V (*t*)

Tabla 3.1. Relación del valor con la salida y la fracción del período del ciclo de trabajo del PWM.

El valor de salida varía de 0 a 5 voltios según el valor de entrada (de 0 a 255) en función del tiempo de pulso. Si *t* es el tiempo de pulso, la tabla 3.1 muestra la equivalencia entre el valor y la salida en función del tiempo.

Ejercicio 3.4:

Determinar la función del siguiente programa.

```
int led = 10;

int pin = 0;

int value;
```

```
void setup (){}
void loop()
{
    value = analogRead(pin);
    value /= 4;
    analogWrite(led, value);
}
```

3.16 Tiempo

3.16.1 delay(ms)

Pausa el programa por la cantidad de tiempo especificado en milisegundos, donde 1000 es igual a 1 segundo.

```
delay(250);    //pausa por un cuarto de segundo el programa
delay(1000);   //pausa por un segundo el programa
```

3.16.2 millis()

Devuelve el número de milisegundos desde que la placa Arduino empezó a ejecutar el programa actual como un valor *long* sin signo.

```
value = millis();     //le asigna a "value" el valor que devuelve la función millis()
```

Nota: El valor que devuelve millis() se desbordará cuando el número de milisegundos sea mayor al que puede almacenar en memoria ese tipo de dato. Cuando esto ocurra, volverá a 0 y comenzará la cuenta nuevamente. El valor de millis() desborda luego de 50 días aproximadamente.

3.17 Matemáticas

3.17.1 min(x,y)

Esta función calcula el mínimo de dos números de cualquier tipo de datos y devuelve el más pequeño.

```
value = min(k,100);   //asigna a "value" el mínimo entre "k" y 100
```

3.17.2 max(x,y)

Esta función calcula el máximo de dos números de cualquier tipo de datos y devuelve el más grande.

```
value = max(k,100);   //asigna a “value” el máximo entre “k” y 100
```

3.18 Serie

3.18.1 Serial.begin(rate)

Esta función abre el puerto serie y asigna la tasa de baudios para la transmisión de datos en serie. La típica tasa de baudios para comunicarse con el ordenador es 9600, aunque otras velocidades también están soportadas.

```
void setup ()
{
        Serial.begin(9600);    //abre el puerto serie y ajusta la tasa de datos a 9600 bps
}
```

Nota: Cuando se usa la comunicación serie, los pines digitales 0 (RX) y 1 (TX) no pueden ser usados al mismo tiempo.

3.18.2 Serial.println(data)

Imprime datos al puerto serie, seguido de un retorno de carro y avance de línea automáticos. Este comando toma la misma forma que *Serial.print()*, pero es más fácil para leer datos en el monitor serial.

Ejemplo 3.8:

Enviar por comunicación serie el valor de una entrada analógica.

```
void setup ()
{
        Serial.begin(9600);        //inicia la conexión serie y ajusta la tasa de transferencia a 9600
}
void loop()
{
        Serial.println(analogRead(0));      //envía el valor analógico de la entrada A0
        delay(1000);      //pausa por un segundo
}
```

Nota 1: Observar que para utilizar esta función, primero hay que iniciarse la conexión serie con la función *Serial.begin()*.

Nota 2: Suele utilizarse un *delay*() luego de enviar datos por el puerto serie con esta función debido a que si se quisiese observarlos en el monitor serial, éste se actualizaría muy rápido sin poder apreciar los datos en él. Por esto, debe pausarse por un lapso de tiempo correspondiente.