ARDUINO DOCUMENTACIÓN OFICIAL

Índice

1. Estructura (I)

1.1. Sketch

1.1.1. setup()

Descripción:

La función `setup()` se llama cuando se inicia un boceto. Úselo para inicializar variables, modos pin, comenzar a usar bibliotecas, etc. La función `setup()` solo se ejecutará una vez, después de cada encendido o reinicio de la placa Arduino.

Código de ejemplo:

```
int buttonPin = 3;
void setup()
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}
void loop() {
  // …
}
```

1.1.2. loop()

Descripción:

Después de crear una función `setup()`, que inicializa y establece los valores iniciales, la función `loop()` hace exactamente lo que sugiere su nombre y se repite consecutivamente, lo que permite que su programa cambie y responda. Úselo para controlar activamente la placa Arduino.

Código de ejemplo:

```
int buttonPin = 3;
// la configuración inicializa el serial y el pin del botón
void setup() {
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}
// loop comprueba el pin del botón cada vez
// y enviará serial si se presiona
void loop()
  if (digitalRead(buttonPin) == HIGH) {
    Serial.write('H');
  }
  else {
    Serial.write('L');
  }
  delay(1000);
}
```

1.2. Sintaxis Adicional

1.2.1. #define (define)

`#define` es un componente útil de C++ que permite al programador dar un nombre a un valor constante antes de compilar el programa. Las constantes definidas en arduino no ocupan ningún espacio de memoria de programa en el chip. El compilador reemplazará las referencias a estas constantes con el valor definido en el momento de la compilación.

Sin embargo, esto puede tener algunos efectos secundarios no deseados, si por ejemplo, un nombre de constante que ha sido #definido se incluye en algún otro nombre de constante o variable. En ese caso, el texto sería reemplazado por el número (o texto) #definido.

En general, se prefiere la palabra clave `const` para definir constantes y debe usarse en lugar de `#define`.

Sintaxis:

```
#define constantName value
```

Parámetros:

`constantName`: el nombre de la macro a definir.
`value`: el valor a asignar a la macro.

Código de ejemplo:

```
#define ledPin 3
// El compilador reemplazará cualquier mención de ledPin con el valor 3
// en el momento de la compilación.
```

Notas y Advertencias:

No hay punto y coma después de la instrucción `#define`. Si incluye uno, el compilador arrojará errores crípticos más abajo en la página.

```
#define ledPin 3; // esto es un error
```

De manera similar, incluir un signo igual después de la declaración #define también generará un error de compilación críptico más abajo en la página.

```
#define ledPin  = 3 // esto también es un error
```

1.2.2. #include(include)

Descripción:

`#include` se usa para incluir bibliotecas externas en su boceto. Esto le da al programador acceso a un gran grupo de bibliotecas C estándar (grupos de funciones prefabricadas), y también bibliotecas escritas especialmente para Arduino.

La página de referencia principal para las bibliotecas AVR C (AVR es una referencia a los chips Atmel en los que se basa Arduino) está aquí (https://www.nongnu.org/avr-libc/user-manual/modules.html).

Tenga en cuenta que `#include`, similar a `#define`, no tiene terminador de punto y coma, y el compilador generará mensajes de error crípticos si agrega uno.

Sintaxis:

```
#include <LibraryFile.h>
#include "LocalFile.h"
```

Parámetros:

`LibraryFile.h`: cuando se usa la sintaxis de corchetes angulares, se buscará el archivo en las rutas de las bibliotecas.
`LocalFile.h`: cuando se usa la sintaxis de comillas dobles, la carpeta del archivo que usa la directiva `#include` se buscará para el archivo especificado, luego las rutas de las bibliotecas si no se encuentra en la ruta local. Utilice esta sintaxis para archivos de encabezado en la carpeta del boceto.

Código de ejemplo:

Este ejemplo incluye la biblioteca Servo para que sus funciones puedan usarse para controlar un motor Servo.

```
#include <Servo.h>

Servo myservo;  // crear objeto servo para controlar un servo

void setup() {
  myservo.attach(9);  // conecta el servo en el pin 9 al objeto servo
}
void loop() {
  for (int pos = 0; pos <= 180; pos += 1)
    {// va de 0 grados a 180 grados en pasos de 1 grado
    myservo.write(pos);  // le dice al servo que vaya a la
    // posición en la variable 'pos'
    delay(15); // espera 15 ms para que el servo alcance la posición
  }
  for (int pos = 180; pos >= 0; pos -= 1){//va de 180 grados a 0 grados
    myservo.write(pos);  // decirle al servo que vaya a la posición en la
    // variable 'pos
    delay(15);   // espera 15 ms para que el servo alcance la posición
  }
}
```

1.2.3. // (comentario de una sola línea)

Descripción:

Los **comentarios** son líneas en el programa que se utilizan para informarse a sí mismo o a otros sobre la forma en que funciona el programa. El compilador los ignora y no los exporta al procesador, por lo que no ocupan espacio en la memoria flash del microcontrolador. El único propósito de los comentarios es ayudarlo a comprender (o recordar) o informar a otros sobre cómo funciona su programa.

Un **comentario de una sola línea** comienza con `//` (dos barras inclinadas adyacentes). Este comentario termina automáticamente al final de una línea. Lo que siga `//` hasta el final de una línea será ignorado por el compilador.

Código de ejemplo:

Hay dos formas diferentes de marcar una línea como comentario:

```
// El pin 13 tiene un LED conectado en la mayoría de las placas Arduino.
// Dale un nombre:
int led = 13;
digitalWrite(led, HIGH);  // encienda el LED (HIGH es el nivel de voltaje)
```

Notas y Advertencias:

Al experimentar con el código, "comentar" partes de su programa es una forma conveniente de eliminar líneas que pueden tener errores. Esto deja las líneas en el código, pero las convierte en comentarios, por lo que el compilador las ignora. Esto puede ser especialmente útil cuando se trata de localizar un problema o cuando un programa se niega a compilar y el error del compilador es críptico o inútil.

1.2.4. /* */ (comentario de bloque)

Descripción:

Los **comentarios** son líneas en el programa que se utilizan para informarse a sí mismo oa otros sobre la forma en que funciona el programa. El compilador los ignora y no los exporta al procesador, por lo que no ocupan espacio en la memoria flash del microcontrolador. El único propósito de los comentarios es ayudarlo a comprender (o recordar), o informar a otros sobre cómo funciona su programa.

El comienzo de un **comentario de bloque** o un **comentario de varias líneas** se marca con el símbolo `/*` y el símbolo `*/` marca su final. Este tipo de comentario se llama así porque puede extenderse por más de una línea; una vez que el compilador lee `/*` ignora lo que sigue hasta que encuentra un `*/`.

Código de ejemplo:

```
/* Este es un comentario valido */
/*
  Blink
  Enciende un LED durante un segundo, luego se apaga durante un segundo,
  repetidamente
  Este código de ejemplo es de dominio público.
  (Otro comentario válido)
*/
/*
  if (gwb == 0) { // el comentario de una sola línea está bien dentro de un
  // comentario de varias líneas
    x = 3; /* pero no otro comentario de varias líneas: esto no es válido*/
  }
// no olvide el comentario de "cierre": ¡tienen que estar equilibrados!
*/
```

Notas y Advertencias:

Al experimentar con el código, "comentar" partes de su programa es una forma conveniente de eliminar líneas que pueden tener errores. Esto deja las líneas en el código, pero las convierte en comentarios, por lo que el compilador las ignora. Esto puede ser especialmente útil cuando se trata de localizar un problema o cuando un programa se niega a compilar y el error del compilador es críptico o inútil.

1.2.5. ; (punto y coma)

Descripción:

Se utiliza para finalizar una declaración.

Código de ejemplo:

```
int a = 13;
```

Notas y Advertencias:

Olvidar terminar una línea con un punto y coma resultará en un error de compilación. El texto de error puede ser obvio y hacer referencia a un punto y coma que falta, o puede que no. Si aparece un error de compilación impenetrable o aparentemente ilógico, una de las primeras cosas que debe verificar es un punto y coma faltante, en las inmediaciones, que precede a la línea en la que se quejó el compilador.

1.2.6. {} (llaves)

Descripción:

Las llaves (también denominadas simplemente "llaves" o "corchetes") son una parte importante del lenguaje de programación C++. Se utilizan en varias construcciones diferentes, que se describen a continuación, y esto a veces puede resultar confuso para los principiantes.

Una llave de apertura `{` debe ir siempre seguida de una llave de cierre `}`. Esta es una condición a la que a menudo se hace referencia como que los frenos están equilibrados. El IDE (Entorno de desarrollo integrado) de Arduino incluye una característica conveniente para comprobar el equilibrio de las llaves. Simplemente seleccione una llave, o incluso haga clic en el punto de inserción inmediatamente después de una llave, y se resaltará su compañero lógico.

Los programadores principiantes y los programadores que llegan a C++ desde el lenguaje BASIC a menudo encuentran confuso o desalentador el uso de llaves. Después de todo, las mismas llaves reemplazan la instrucción RETURN en una subrutina (función), la instrucción ENDIF en un condicional y la instrucción NEXT en un bucle FOR.

Las llaves desequilibradas a menudo pueden conducir a errores de compilación crípticos e impenetrables que a veces pueden ser difíciles de rastrear en un programa grande. Debido a sus variados usos, las llaves también son increíblemente importantes para la sintaxis de un programa y mover una llave una o dos líneas a menudo afectará dramáticamente el significado de un programa.

Código de ejemplo:

Los principales usos de las llaves se enumeran en los ejemplos a continuación.

Funciones

```
void myfunction(argumento de tipo de datos) {
  // cualquier afirmación
}
```

Loops

```
while (expresión booleana) {
  // cualquier afirmación
}

do {
  // cualquier afirmación
} while (expresión booleana);

for (inicialización; condición de terminación; incrementando expr) {
  // cualquier afirmación
}
```

Declaraciones condicionales

```
if (expresión booleana) {
  // cualquier afirmación
}
else if (expresión booleana) {
  // cualquier afirmación
}
else {
  // cualquier afirmación
}
```

2. Funciones

2.1. I/O Digital

2.1.1. pinMode()

Descripción:

Configura el pin especificado para que se comporte como una entrada o una salida. Consulte la página Pines digitales para obtener detalles sobre la funcionalidad de los pines.

A partir de Arduino 1.0.1, es posible habilitar las resistencias pullup internas con el modo `INPUT_PULLUP`. Además, el modo `INPUT` deshabilita explícitamente los pullups internos.

Sintaxis:

```
pinMode(pin, mode)
```

Parámetros:

`pin`: el número de pin de Arduino para establecer el tipo de modo.
`mode`: `INPUT`, `OUTPUT`, o `INPUT_PULLUP`. Consulte la página Pines digitales para obtener una descripción más completa de la funcionalidad.

Devuelve:

Nada

Código de ejemplo:

El código hace que el pin digital 13 sea `OUTPUT` y lo alterna `HIGH` y `LOW`.

```
void setup() {
  pinMode(13, OUTPUT);    // establece el pin digital 13 como salida
}
void loop() {
  digitalWrite(13, HIGH); // activa el pin digital 13
  delay(1000);            // espera un segundo
  digitalWrite(13, LOW);  // desactiva el pin digital 13
  delay(1000);            // espera un segundo
}
```

Notas y Advertencias:

Los pines de entrada analógica se pueden usar como pines digitales, denominados A0, A1, etc.

2.1.2. digitalRead()

Descripción:

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

Sintaxis:

```
digitalRead(pin)
```

Parámetros:

`pin`: el número de pin de Arduino que desea leer

Devuelve:

`HIGH` o `LOW`

Código de ejemplo:

Establece el pin 13 al mismo valor que el pin 7, declarado como entrada.

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

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

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

Notas y Advertencias:

Si el pin no está conectado a nada, `digitalRead()` puede devolver `HIGH` o `LOW` (y esto puede cambiar aleatoriamente).

Los pines de entrada analógica se pueden usar como pines digitales, denominados A0, A1, etc. La excepción son los pines A6 y A7 de Arduino Nano, Pro Mini y Mini, que solo se pueden usar como entradas analógicas.

2.1.3. digitalWrite()

Descripción:

Escriba un valor `HIGH` o `LOW` en un pin digital.

Si el pin ha sido configurado como `OUTPUT` con `pinMode()`, su voltaje se establecerá en el valor correspondiente: 5V (o 3.3V en placas de 3.3V) para `HIGH`, 0V (tierra) para `LOW`.

Si el pin está configurado como `INPUT`, `digitalWrite()` habilitará (`HIGH`) o deshabilitará (`BAJO`) el pullup interno en el pin de entrada. Se recomienda establecer `pinMode()` en `INPUT_PULLUP` para habilitar la resistencia pull-up interna. Consulte el tutorial Pines digitales para obtener más información.

Si no configura `pinMode()` en `OUTPUT` y conecta un LED a un pin, cuando llame a `digitalWrite(HIGH)`, el LED puede aparecer tenue. Sin establecer explícitamente `pinMode()`, `digitalWrite()` habrá habilitado la resistencia pull-up interna, que actúa como una gran resistencia limitadora de corriente.

Sintaxis:

```
digitalWrite(pin, value)
```

Parámetros:

`pin`: el pin de arduino.
`value`: `HIGH` o `LOW`.

Devuelve:

Nada.

Código de ejemplo:

El código convierte el pin digital 13 en una SALIDA y lo cambia alternando entre ALTO y BAJO a un ritmo de un segundo.

```
void setup()
  pinMode(13, OUTPUT);    // establece el pin digital 13 como salida
}

void loop() {
  digitalWrite(13, HIGH); // activa el pin digital 13
  delay(1000);            // espera un segundo
  digitalWrite(13, LOW);  // desactiva el pin digital 13
  delay(1000);            // espera un segundo
}
```

Notas y Advertencias:

Los pines de entrada analógica se pueden usar como pines digitales, denominados A0, A1, etc. La excepción son los pines A6 y A7 de Arduino Nano, Pro Mini y Mini, que solo se pueden usar como entradas analógicas.

2.2. I/O Analógico

2.2.1. analogRead()

Descripción:

Lee el valor del pin analógico especificado. Las placas Arduino contienen un convertidor analógico a digital multicanal de 10 bits. Esto significa que mapeará los voltajes de entrada entre 0 y el voltaje de operación (5V o 3.3V) en valores enteros entre 0 y 1023. En un Arduino UNO, por ejemplo, esto produce una resolución entre lecturas de: 5 voltios / 1024 unidades o , 0,0049 voltios (4,9 mV) por unidad. Consulte la tabla a continuación para conocer los pines utilizables, el voltaje de funcionamiento y la resolución máxima para algunas placas Arduino.

El rango de entrada se puede cambiar usando `analogReference()`, mientras que la resolución se puede cambiar (solo para tarjetas Zero, Due y MKR) usando `analogReadResolution()`.

En las placas basadas en ATmega (UNO, Nano, Mini, Mega), se tarda unos 100 microsegundos (0,0001 s) en leer una entrada analógica, por lo que la tasa de lectura máxima es de unas 10.000 veces por segundo.

PLACA	TENSIÓN DE FUNCIONAMIENTO	PINES UTILIZABLES	RESOLUCIÓN MÁXIMA
Uno	5 Volts	A0 to A5	10 bits
Mini, Nano	5 Volts	A0 to A7	10 bits
Mega, Mega2560, MegaADK	5 Volts	A0 to A14	10 bits
Micro	5 Volts	A0 to A11*	10 bits
Leonardo	5 Volts	A0 to A11*	10 bits
Zero	3.3 Volts	A0 to A5	12 bits**
Due	3.3 Volts	A0 to A11	12 bits**
MKR Family boards	3.3 Volts	A0 to A6	12 bits**

*A0 a A5 están etiquetados en la placa, A6 a A11 están disponibles respectivamente en los pines 4, 6, 8, 9, 10 y 12
**La resolución predeterminada de `analogRead()` para estas placas es de 10 bits, por motivos de compatibilidad. Debe usar `analogReadResolution()` para cambiarlo a 12 bits.

Sintaxis:

```
analogRead(pin)
```

Parámetros:

`pin`: el nombre del pin de entrada analógica para leer (A0 a A5 en la mayoría de las placas, A0 a A6 en las placas MKR, A0 a A7 en Mini y Nano, A0 a A15 en Mega)

Devuelve:

La lectura analógica en el pin. Aunque está limitado a la resolución del conversor analógico a digital (0-1023 para 10 bits o 0-4095 para 12 bits). Tipo de dato: `int`.

Código de ejemplo:

El código lee el voltaje en analogPin y lo muestra.

```
int analogPin = A3; // potenciómetro de cursor (Terminal del medio)
  // Conectado al pin analógico 3 cables externos a tierra y +5V
int val = 0;  // variable para almacenar el valor leído

void setup() {
  Serial.begin(9600);           //  configuración serial
}

void loop() {
  val = analogRead(analogPin);  // leer el pin de entrada
  Serial.println(val);          // valor de depuración
}
```

Notas y Advertencias:

Si el pin de entrada analógica no está conectado a nada, el valor devuelto por `analogRead()` fluctuará en función de una serie de factores (por ejemplo, los valores de las otras entradas analógicas, qué tan cerca está su mano del tablero, etc.).

2.2.2. analogReference()

Descripción:

Configura el voltaje de referencia utilizado para la entrada analógica (es decir, el valor utilizado como la parte superior del rango de entrada). Las opciones son:

Placas Arduino AVR (Uno, Mega, Leonardo, etc.)

- DEFAULT: la referencia analógica predeterminada de 5 voltios (en placas Arduino de 5 V) o 3,3 voltios (en placas Arduino de 3,3 V)
- INTERNO: una referencia incorporada, igual a 1,1 voltios en ATmega168 o ATmega328P y 2,56 voltios en ATmega32U4 y ATmega8 (no disponible en Arduino Mega)
- INTERNAL1V1: una referencia de 1.1V incorporada (solo Arduino Mega)
- INTERNAL2V56: una referencia integrada de 2,56 V (solo Arduino Mega)
- EXTERNO: el voltaje aplicado al pin AREF (solo 0 a 5V) se usa como referencia.

Placas Arduino SAMD (Cero, etc.)

- AR_DEFAULT: la referencia analógica predeterminada de 3,3 V
- AR_INTERNAL: una referencia de 2,23 V incorporada
- AR_INTERNAL1V0: una referencia de 1,0 V incorporada
- AR_INTERNAL1V65: una referencia de 1,65 V incorporada
- AR_INTERNAL2V23: una referencia de 2,23 V incorporada
- AR_EXTERNAL: el voltaje aplicado al pin AREF se usa como referencia

Placas Arduino megaAVR (Uno WiFi Rev2)

- DEFAULT: una referencia de 0,55 V incorporada
- INTERNAL: una referencia de 0,55 V incorporada
- VDD: Vdd del ATmega4809. 5V en el Uno WiFi Rev2
- INTERNAL0V55: una referencia de 0,55 V incorporada
- INTERNAL1V1: una referencia de 1,1 V integrada
- INTERNAL1V5: una referencia de 1,5 V integrada
- INTERNAL2V5: una referencia de 2,5 V integrada
- INTERNAL4V3: una referencia de 4,3 V incorporada
- EXTERNO: el voltaje aplicado al pin AREF (solo 0 a 5V) se usa como referencia

Placas Arduino SAM (por vencimiento)

- AR_DEFAULT: la referencia analógica predeterminada de 3,3 V. Esta es la única opción admitida para el vencimiento.

Arduino Mbed OS Nano Boards (Nano 33 BLE), Arduino Mbed OS Edge Boards (Edge Control)

- AR_VDD: la referencia predeterminada de 3,3 V
- AR_INTERNAL: referencia de 0,6 V incorporada

- AR_INTERNAL1V2: referencia de 1,2 V (referencia interna de 0,6 V con ganancia 2x)
- AR_INTERNAL2V4: referencia de 2,4 V (referencia interna de 0,6 V con ganancia 4x)

Sintaxis:

```
analogReference(type)
```

Parámetros:

`type`: qué tipo de referencia usar (ver lista de opciones en la descripción).

Devuelve:

Nada

Notas y Advertencias:

Después de cambiar la referencia analógica, las primeras lecturas de `analogRead()` pueden no ser precisas.

¡No use menos de 0 V ni más de 5 V para el voltaje de referencia externo en el pin AREF! Si está utilizando una referencia externa en el pin AREF, debe establecer la referencia analógica en EXTERNA antes de llamar a `analogRead()`. De lo contrario, cortocircuitará el voltaje de referencia activo (generado internamente) y el pin AREF, posiblemente dañando el microcontrolador en su placa Arduino.

Alternativamente, puede conectar el voltaje de referencia externo al pin AREF a través de una resistencia de 5K, lo que le permite cambiar entre voltajes de referencia internos y externos. Tenga en cuenta que la resistencia alterará el voltaje que se usa como referencia porque hay una resistencia interna de 32K en el pin AREF. Los dos actúan como un divisor de voltaje, por lo que, por ejemplo, 2,5 V aplicados a través de la resistencia producirán 2,5 * 32 / (32 + 5) = ~2,2 V en el pin AREF.

2.2.3. analogWrite()

Descripción:

Escribe un valor analógico (onda PWM) en un pin. Se puede usar para encender un LED con diferentes brillos o impulsar un motor a varias velocidades. Después de una llamada a `analogWrite()`, el pin generará una onda rectangular constante del ciclo de trabajo especificado hasta la próxima llamada a `analogWrite()` (o una llamada a `digitalRead()` o `digitalWrite()`) en el mismo pin.

BOARD	PWM PINS	PWM FREQUENCY
Uno, Nano, Mini	3, 5, 6, 9, 10, 11	490 Hz (pins 5 and 6: 980 Hz)
Mega	2 - 13, 44 - 46	490 Hz (pins 4 and 13: 980 Hz)
Leonardo, Micro, Yún	3, 5, 6, 9, 10, 11, 13	490 Hz (pins 3 and 11: 980 Hz)
Uno WiFi Rev2, Nano Every	3, 5, 6, 9, 10	976 Hz
MKR boards *	0 - 8, 10, A3, A4	732 Hz
MKR1000 WiFi *	0 - 8, 10, 11, A3, A4	732 Hz
Zero *	3 - 13, A0, A1	732 Hz

BOARD	PWM PINS	PWM FREQUENCY
Nano 33 IoT *	2, 3, 5, 6, 9 - 12, A2, A3, A5	732 Hz
Nano 33 BLE/BLE Sense	1 - 13, A0 - A7	500 Hz
Due **	2-13	1000 Hz
101	3, 5, 6, 9	pins 3 and 9: 490 Hz, pins 5 and 6: 980 Hz

* Además de las capacidades de PWM en los pines indicados anteriormente, las placas MKR, Nano 33 IoT y Zero tienen salida analógica verdadera cuando se usa `analogWrite()` en el pin `DAC0(A0)`.
** Además de las capacidades de PWM en los pines mencionados anteriormente, Due tiene una salida analógica verdadera cuando se usa `analogWrite()` en los pines `DAC0` y `DAC1`.

No necesita llamar a `pinMode()` para configurar el pin como salida antes de llamar a `analogWrite()`. La función `analogWrite` no tiene nada que ver con los pines analógicos o la función `analogRead`.

Sintaxis:

```
analogWrite(pin, value)
```

Parámetros:

`pin`: el pin de Arduino para escribir. Tipos de datos permitidos: `int`.
`value`: el ciclo de trabajo: entre 0 (siempre apagado) y 255 (siempre encendido). Tipos de datos permitidos: `int`.

Devuelve:

Nada

Código de ejemplo:

Establece la salida al LED proporcional al valor leído del potenciómetro.

```
int ledPin = 9;      // LED conectado al pin digital 9
int analogPin = 3;   // potenciómetro conectado al pin analógico 3
int val = 0;         // variable para almacenar el valor leído

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

void loop() {
  val = analogRead(analogPin);  // leer el pin de entrada
  analogWrite(ledPin, val / 4); // Los valores de lectura analógica
         //van de 0 a 1023, los valores de escritura analógica de 0 a 255
}
```

Notas y Advertencias:

Las salidas PWM generadas en los pines 5 y 6 tendrán ciclos de trabajo más altos de lo esperado. Esto se debe a las interacciones con las funciones `millis()` y `delay()`, que comparten el mismo temporizador interno que se usa para generar esas salidas PWM. Esto se notará principalmente en configuraciones de ciclo de trabajo bajo (por ejemplo, 0 - 10) y puede resultar en un valor de 0 que no apague completamente la salida en los pines 5 y 6.

2.3. Zero, DUe & MKR Family

2.3.1. analogReadResolution()

Descripción:

`analogReadResolution()` es una extensión de *Analog API* para la familia Zero, Due, MKR, Nano 33 (BLE e IoT) y Portenta.

Establece el tamaño (en bits) del valor devuelto por `analogRead()`. Tiene un valor predeterminado de 10 bits (devuelve valores entre 0 y 1023) para compatibilidad con versiones anteriores de placas basadas en AVR.

Las placas de la familia **Zero, Due, MKR y Nano 33 (BLE e IoT)** tienen capacidades ADC de 12 bits a las que se puede acceder cambiando la resolución a 12. Esto devolverá valores de `analogRead()` entre 0 y 4095. El **Portenta H7** tiene un ADC de 16 bits, que permitirá valores entre 0 y 65535.

Sintaxis:

```
analogReadResolution(bits)
```

Parámetros:

`bits`: determina la resolución (en bits) del valor devuelto por la función `analogRead()`. Puede establecer esto entre 1 y 32. Puede establecer resoluciones más altas que los 12 o 16 bits admitidos, pero los valores devueltos por `analogRead()` sufrirán una aproximación. Consulte la nota a continuación para obtener más detalles.

Devuelve:

Nada

Código de ejemplo:

El código muestra cómo usar ADC con diferentes resoluciones.

```
void setup() {
// abrir una conexión en serie
     Serial.begin(9600);
}
void loop() {
     // leer la entrada en A0 a la resolución predeterminada (10 bits)
     // y enviarlo por la conexión en serie
     analogReadResolution(10);
     Serial.print("ADC 10-bit (default) : ");
     Serial.print(analogRead(A0));
     // cambie la resolución a 12 bits y lea A0
     analogReadResolution(12);
     Serial.print(", 12-bit : ");
     Serial.print(analogRead(A0));
     // cambie la resolución a 16 bits y lea A0
     analogReadResolution(16);
     Serial.print(", 16-bit : ");
     Serial.print(analogRead(A0));
     // cambie la resolución a 8 bits y lea A0
     analogReadResolution(8);
```

```
    Serial.print(", 8-bit : ");
    Serial.println(analogRead(A0));
    // un poco de retraso para no acaparar Serial Monitor
    delay(100);
}
```

Notas y advertencias:

Si establece el valor `analogReadResolution()` en un valor superior a las capacidades de su placa, el Arduino solo informará a su resolución más alta, rellenando los bits adicionales con ceros.

Por ejemplo: usar Due con `analogReadResolution(16)` le dará un número aproximado de 16 bits con los primeros 12 bits que contienen la lectura real de ADC y los últimos 4 bits **rellenos con ceros.**

Si configura el valor `analogReadResolution()` en un valor inferior a las capacidades de su placa, los bits extra menos significativos leídos del ADC se **descartarán**.

El uso de una resolución de 16 bits (o cualquier resolución **superior** a las capacidades reales del hardware) le permite escribir bocetos que manejen automáticamente dispositivos con un ADC de mayor resolución cuando estén disponibles en placas futuras sin cambiar una línea de código.

2.3.2. analogWriteResolution()

Descripción:

`analogWriteResolution()` es una extensión de la API analógica para Arduino Due.

a`nalogWriteResolution()` establece la resolución de la función `analogWrite()`. Tiene un valor predeterminado de 8 bits (valores entre 0 y 255) para compatibilidad con versiones anteriores de placas basadas en AVR.

El **Due** tiene las siguientes capacidades de hardware:

- 12 pines que por defecto son PWM de 8 bits, como las placas basadas en AVR. Estos se pueden cambiar a una resolución de 12 bits.
- 2 pines con DAC de 12 bits (convertidor de digital a analógico)

Al configurar la resolución de escritura en 12, puede usar `analogWrite()` con valores entre 0 y 4095 para explotar la resolución completa de DAC o para configurar la señal PWM sin cambiar.

El **Zero** tiene las siguientes capacidades de hardware:

- 10 pines que por defecto son PWM de 8 bits, como las placas basadas en AVR. Estos se pueden cambiar a una resolución de 12 bits.
- 1 pin con DAC de 10 bits (convertidor de digital a analógico).

Al configurar la resolución de escritura en 10, puede usar `analogWrite()` con valores entre 0 y 1023 para aprovechar la resolución DAC completa

La **familia de placas MKR** tiene las siguientes capacidades de hardware:

- 4 pines que por defecto son PWM de 8 bits, como las placas basadas en AVR. Estos se pueden cambiar de 8 (predeterminado) a 12 bits de resolución.

- 1 pin con DAC de 10 bits (convertidor de digital a analógico)

Al establecer la resolución de escritura en 12 bits, puede usar `analogWrite()` con valores entre 0 y 4095 para señales PWM; establezca 10 bits en el pin DAC para explotar la resolución DAC completa de 1024 valores.

Sintaxis:

```
analogWriteResolution(bits)
```

Parámetros:

`bits`: determina la resolución (en bits) de los valores utilizados en la función `analogWrite()`. El valor puede oscilar entre 1 y 32. Si elige una resolución superior o inferior a las capacidades de hardware de su placa, el valor utilizado en `analogWrite()` se truncará si es demasiado alto o se completará con ceros si es demasiado bajo. Consulte la nota a continuación para obtener más detalles.

Devuelve:

Nada

Código de ejemplo:

```
void setup() {
  // abrir una conexión en serie
  Serial.begin(9600);
  // convertir nuestro pin digital en una salida
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  // lea la entrada en A0 y mapéela a un pin PWM con un LED adjunto
  int sensorVal = analogRead(A0);
  Serial.print("Analog Read) : ");
  Serial.print(sensorVal);

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

  // cambiar la resolución PWM a 12 bits solo se admite la resolución
  // completa de 12 bits en el Due
  analogWriteResolution(12);
  analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
  Serial.print(" , 12-bit PWM value : ");
  Serial.print(map(sensorVal, 0, 1023, 0, 4095));

  // cambiar la resolución PWM a 4 bits
  analogWriteResolution(4);
  analogWrite(13, map(sensorVal, 0, 1023, 0, 15));
  Serial.print(", 4-bit PWM value : ");
  Serial.println(map(sensorVal, 0, 1023, 0, 15));
  delay(5);
}
```

Notas y Advertencias:

Si establece el valor `analogWriteResolution()` en un valor superior a las capacidades de su placa, el Arduino descartará los bits adicionales. Por ejemplo: al usar Due con `analogWriteResolution(16)` en un pin DAC de 12 bits, solo se usarán los primeros 12 bits de los valores pasados a `analogWrite()` y se descartarán los últimos 4 bits.

Si establece el valor `analogWriteResolution()` en un valor inferior a las capacidades de su placa, los bits faltantes **se rellenarán con ceros** para llenar el tamaño requerido de hardware. Por ejemplo: usando Due con `analogWriteResolution(8)` en un pin DAC de 12 bits, Arduino agregará 4 bits cero al valor de 8 bits usado en `analogWrite()` para obtener los 12 bits requeridos.

2.4. Advanced I/O

2.4.1. tone()

Descripción:

Genera una onda cuadrada de la frecuencia especificada (y un ciclo de trabajo del 50 %) en un pin. Se puede especificar una duración; de lo contrario, la onda continúa hasta que se llama a `noTone()`. El pin se puede conectar a un zumbador piezoeléctrico u otro altavoz para reproducir tonos.

Solo se puede generar un tono a la vez. Si ya se está reproduciendo un tono en un pin diferente, la llamada a `tone()` no tendrá efecto. Si el tono se reproduce en el mismo pin, la llamada establecerá su frecuencia.

El uso de la función `tone()` interferirá con la salida PWM en los pines 3 y 11 (en placas que no sean Mega).

No es posible generar tonos inferiores a 31Hz. Para obtener detalles técnicos, consulte las notas de Brett Hagman (https://github.com/bhagman/Tone#ugly-details).

Sintaxis:

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

Parámetros:

`pin` : el pin Arduino en el que generar el tono.
`frequency`: la frecuencia del tono en hercios. Tipos de datos permitidos: `unsigned int`.
`duration` : la duración del tono en milisegundos (opcional). Tipos de datos permitidos: `unsigned long`.

Devuelve:

Nada

Notas y Advertencias:

Si desea tocar diferentes tonos en varios pines, debe llamar a `noTone()` en un pin antes de llamar a `tone()` en el siguiente pin.

2.4.2. noTone()

Descripción:

Detiene la generación de una onda cuadrada desencadenada por tone(). No tiene efecto si no se genera ningún tono.

Sintaxis:

```
noTone(pin)
```

Parámetros:

`pin`: el pin Arduino en el que dejar de generar el tono

Devuelve:

Nada

Notas y advertencias:

Si desea tocar diferentes tonos en varios pines, debe llamar a `noTone()` en un pin antes de llamar a `tone()` en el siguiente pin.

2.4.3. pulseIn()

Descripción:

Lee un pulso (ya sea `HIGH` o `LOW`) en un pin. Por ejemplo, si el `valor` es `HIGH`, `pulseIn()` espera a que el pin pase de `LOW` a `HIGH`, comienza a cronometrar, luego espera a que el pin pase a `LOW` y detiene el cronometraje. Devuelve la longitud del pulso en microsegundos o se da por vencido y devuelve 0 si no se recibió ningún pulso completo dentro del tiempo de espera.

El tiempo de esta función ha sido determinado empíricamente y probablemente mostrará errores en pulsos más largos. Funciona con pulsos de 10 microsegundos a 3 minutos de duración.

Nota: si se usa el tiempo de espera opcional, el código se ejecutará más rápido.

Sintaxis:

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

Parámetros:

`pin`: el número del pin Arduino en el que desea leer el pulso. Tipos de datos permitidos: `int`.
`value`: tipo de pulso a leer: ALTO o BAJO. Tipos de datos permitidos: `int`.
`timeout` (opcional): el número de microsegundos a esperar para que comience el pulso; el valor predeterminado es un segundo. Tipos de datos permitidos: `unsigned long`.

Devuelve:

La duración del pulso (en microsegundos) o 0 si no se inició ningún pulso antes del tiempo de espera. Tipo de datos: `unsigned long`.

Código de ejemplo:

El ejemplo imprime el tiempo de duración de un pulso en el pin 7

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

void setup() {
  Serial.begin(9600);
  pinMode(pin, INPUT);
}
void loop() {
  duration = pulseIn(pin, HIGH);
  Serial.println(duration);
}
```

2.4.4. pulseInLong()

Descripción:

`pulseInLong()` es una alternativa a `pulseIn()` que es mejor en el manejo de pulsos largos y escenarios afectados por interrupciones.

Lee un pulso (ya sea `HIGH` o `LOW`) en un pin. Por ejemplo, si `value` es `HIGH`, `pulseInLong()` espera a que el pin pase de `LOW` a `HIGH`, comienza a cronometrar, luego espera a que el pin pase a `LOW` y detiene el cronometraje. Devuelve la longitud del pulso en microsegundos o se da por vencido y devuelve 0 si no se recibió ningún pulso completo dentro del tiempo de espera.

El tiempo de esta función ha sido determinado empíricamente y probablemente mostrará errores en pulsos más cortos. Funciona con pulsos de 10 microsegundos a 3 minutos de duración. Esta rutina se puede usar solo si las interrupciones están activadas. Además, la resolución más alta se obtiene con intervalos grandes.

Sintaxis:

```
pulseInLong(pin, value)
pulseInLong(pin, value, timeout)
```

Parámetros:

`pin`: el número del pin Arduino en el que desea leer el pulso. Tipos de datos permitidos: `int`.
`value`: tipo de pulso a leer: ALTO o BAJO. Tipos de datos permitidos: `int`.
`timeout` (opcional): el número de microsegundos a esperar para que comience el pulso; el valor predeterminado es un segundo. Tipos de datos permitidos: `unsigned long`.

Devuelve:

La duración del pulso (en microsegundos) o 0 si no se inició ningún pulso antes del tiempo de espera. Tipo de datos: `unsigned long`.

Código de ejemplo:

El ejemplo imprime el tiempo de duración de un pulso en el pin 7.

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

void setup() {
  Serial.begin(9600);
  pinMode(pin, INPUT);
}
void loop() {
  duration = pulseInLong(pin, HIGH);
  Serial.println(duration);
}
```

Notas y advertencias:

Esta función se basa en `micros()`, por lo que no se puede usar en el contexto `noInterrupts()`.

2.4.5. shiftIn()

Descripción:

Shifts en un byte de datos un bit a la vez. Comienza desde el bit más significativo (es decir, el más a la izquierda) o menos (el más a la derecha). Para cada bit, el pin del reloj se eleva, el siguiente bit se lee de la línea de datos y luego el pin del reloj se baja.

Si está interactuando con un dispositivo que tiene un reloj con flancos ascendentes, deberá asegurarse de que el pin del reloj esté bajo antes de la primera llamada a `shiftIn()`, ej. con una llamada a `digitalWrite(clockPin, LOW)`.

Nota: esta es una implementación de software; Arduino también proporciona una biblioteca SPI que utiliza la implementación de hardware, que es más rápida pero solo funciona en pines específicos.

Sintaxis:

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

Parámetros:

`dataPin`: el pin en el que ingresar cada bit. Tipos de datos permitidos: `int`.
`clockPin`: el pin para alternar para señalar una lectura de **dataPin**.
`bitOrder`: qué orden cambiar en los bits; ya sea **MSBFIRST** o **LSBFIRST**. (Primero el bit más significativo o Primero el bit menos significativo).

Devuelve:

El valor leído. Tipo de dato: `byte`.

2.4.6. shiftOut()

Descripción:

Desplaza un byte de datos un bit a la vez. Comienza desde el bit más significativo (es decir, el más a la izquierda) o menos (el más a la derecha). Cada bit se escribe a su vez en un pin de datos, después de lo cual se pulsa un pin de reloj (se toma alto, luego bajo) para indicar que el bit está disponible.

Nota: si está interactuando con un dispositivo que tiene un reloj con flancos ascendentes, deberá asegurarse de que el pin del reloj esté bajo antes de la llamada a `shiftOut()`, ej. con una llamada a `digitalWrite(clockPin, LOW)`.

Esta es una implementación de software; consulte también la biblioteca SPI, que proporciona una implementación de hardware que es más rápida pero funciona solo en pines específicos.

Sintaxis:

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

Parámetros:

`dataPin`: el pin en el que generar cada bit. Tipos de datos permitidos: `int`.
`clockPin`: el pin para alternar una vez que el pin de datos se haya establecido en el valor correcto. Tipos de datos permitidos: `int`.
`bitOrder`: en qué orden desplazar los bits; ya sea MSBFIRST o LSBFIRST. (Primero el bit más significativo o Primero el bit menos significativo).
`valor`: los datos a desplazar. Tipos de datos permitidos: `byte`.

Devuelve:

Nada.

Código de ejemplo:

```
//*****************************************************//
// Notes : Codigo para usar el 74HC595 Shift Register   //
//            : para contar desde 0 a 255               //
//*****************************************************//

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

void setup() {
// establezca los pines en la salida porque están direccionados en el
// bucle principal
     pinMode(latchPin, OUTPUT);
     pinMode(clockPin, OUTPUT);
     pinMode(dataPin, OUTPUT);
}
```

```
void loop() {
//rutina de conteo
  for (int j = 0; j < 256; j++) {
    //Pin del pestillo de tierra y manténgalo presionado mientras
    //esté transmitiendo
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    //devuelva el pasador del pestillo alto para indicarle al chip que
    // ya no necesita escucha información
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

Notas y Advertencias:

El pin de datos y el pin de reloj ya deben estar configurados como salidas mediante una llamada a `pinMode()`.

`shiftOut` actualmente está escrito para generar 1 byte (8 bits), por lo que requiere una operación de dos pasos para generar valores mayores que 255.

```
// Haga esto para la serie MSBFIRST
int data = 500;
// desplazar byte alto
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// cambiar byte bajo
shiftOut(dataPin, clock, MSBFIRST, data);
// O haga esto para la serie LSBFIRST
data = 500;
// shift out lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// desplazar byte alto
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

2.5. Tiempo

2.5.1. delay()

Descripción:

Pausa el programa por la cantidad de tiempo (en milisegundos) especificado como parámetro. (Hay 1000 milisegundos en un segundo.)

Sintaxis:

```
delay(ms)
```

Parámetros:

`ms`: el número de milisegundos para hacer una pausa. Tipos de datos permitidos: `unsigned long`.

Devuelve:

Nada

Código de ejemplo:

El código detiene el programa durante un segundo antes de alternar el pin de salida.

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

void setup() {
  pinMode(ledPin, OUTPUT); // establece el pin digital como salida
}
void loop() {
  digitalWrite(ledPin, HIGH); // senciende el LED
  delay(1000); // espera un segundo
  digitalWrite(ledPin, LOW); // apaga el LED
  delay(1000); // espera un segundo
}
```

Notas y Advertencias:

Si bien es fácil crear un LED parpadeante con la función de `delay()` y muchos bocetos usan retrasos breves para tareas como la eliminación de rebotes de interruptores, el uso de `delay()` en un sketch tiene importantes inconvenientes. Ninguna otra lectura de sensores, cálculos matemáticos o manipulación de pines puede continuar durante la función de retraso, por lo que, en efecto, detiene la mayoría de las demás actividades. Para obtener enfoques alternativos para controlar el tiempo, consulte el sketch Blink Without Delay, que realiza un loop, sondeando la función `millis()` hasta que haya transcurrido suficiente tiempo. Los programadores con más conocimientos generalmente evitan el uso de `delay()` para cronometrar eventos de más de 10 milisegundos, a menos que el boceto de Arduino sea muy simple.

Sin embargo, ciertas cosas continúan mientras la función de `delay()` controla el chip Atmega, porque la función de retraso no desactiva las interrupciones. Se registra la comunicación en serie que aparece en el pin RX, se mantienen los valores PWM (`analogWrite`) y los estados del pin, y las interrupciones funcionarán como deberían.

2.5.2. delayMicroseconds()

Descripción:

Pausa el programa por la cantidad de tiempo (en microsegundos) especificado por el parámetro. Hay mil microsegundos en un milisegundo y un millón de microsegundos en un segundo.

Actualmente, el valor más grande que producirá un retraso preciso es 16383; valores más grandes pueden producir un retardo extremadamente corto. Esto podría cambiar en futuras versiones de Arduino. Para retrasos de más de unos pocos miles de microsegundos, debe usar `delay()` en su lugar.

Sintaxis:

```
delayMicroseconds(us)
```

Parámetros:

us: el número de microsegundos para hacer una pausa. Tipos de datos permitidos: `unsigned int`.

Devuelve:

Nada

Código de ejemplo:

El código configura el pin número 8 para que funcione como un pin de salida. Envía un tren de pulsos de aproximadamente 100 microsegundos de período. La aproximación se debe a la ejecución de las otras instrucciones en el código.

```
int outPin = 8;               // digital pin 8

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

void loop() {
  digitalWrite(outPin, HIGH); // pone el pin en ON
  delayMicroseconds(50);      // pausas de 50 microsegundos
  digitalWrite(outPin, LOW);  // pone el pin en OFF
  delayMicroseconds(50);      // pausas de 50 microsegundos
}
```

Notas y Advertencias:

Esta función funciona con mucha precisión en el rango de 3 microsegundos y hasta 16383. No podemos asegurar que los microsegundos de retardo funcionen con precisión para tiempos de retardo más pequeños. Los tiempos de retardo más grandes en realidad pueden retrasar por un tiempo extremadamente breve.

A partir de Arduino 0018, delayMicroseconds() ya no desactiva las interrupciones.

2.5.3. micros()

Descripción:

Devuelve el número de microsegundos desde que la placa Arduino comenzó a ejecutar el programa actual. Este número se desbordará (volverá a cero) después de aproximadamente 70 minutos. En las placas de la familia Arduino Portenta esta función tiene una resolución de un microsegundo en todos los núcleos. En placas Arduino de 16 MHz (por ejemplo, Duemilanove y Nano), esta función tiene una resolución de cuatro microsegundos (es decir, el valor devuelto es siempre un múltiplo de cuatro). En placas Arduino de 8 MHz (por ejemplo, LilyPad), esta función tiene una resolución de ocho microsegundos.

Sintaxis:

```
time = micros()
```

Parámetros:

Ninguno

Devuelve:

Devuelve el número de microsegundos desde que la placa Arduino comenzó a ejecutar el programa actual. Tipo de datos: `unsigned long`.

Código de ejemplo:

El código devuelve el número de microsegundos desde que comenzó la placa Arduino.

```
unsigned long time;
void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Time: ");
  time = micros();

  Serial.println(time); //imprime el tiempo desde que se inició el
  // programa
  delay(1000);        // espera un segundo para no enviar cantidades
  // masivas de datos
}
```

Notas y Advertencias:

Hay 1,000 microsegundos en un milisegundo y 1,000,000 de microsegundos en un segundo.

2.5.4. millis()

Descripción:

Devuelve el número de milisegundos transcurridos desde que la placa Arduino comenzó a ejecutar el programa actual. Este número se desbordará (volverá a cero), después de aproximadamente 50 días.

Sintaxis:

```
time = millis()
```

Parámetros:

Ninguno

Devuelve:

Número de milisegundos transcurridos desde que se inició el programa. Tipo de datos: `unsigned long`.

Código de ejemplo:

Este código de ejemplo imprime en el puerto serie la cantidad de milisegundos transcurridos desde que la placa Arduino comenzó a ejecutar el código.

```
unsigned long myTime;

void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Time: ");
  myTime = millis();
```

```
  Serial.println(myTime); // imprime el tiempo desde que se inició el
                          //programa
  delay(1000); // imprime el tiempo desde que se inicio el programa
}
```

Notas y Advertencias:

Tenga en cuenta que el valor de retorno para `millis()` es de tipo `unsigned long`, pueden ocurrir errores lógicos si un programador intenta hacer aritmética con tipos de datos más pequeños como `int`. Incluso firmado largo puede encontrar errores ya que su valor máximo es la mitad del de su contraparte sin firmar.

La reconfiguración de los temporizadores del microcontrolador puede generar lecturas de `millis()` imprecisas. Los núcleos "Arduino AVR Boards" y "Arduino megaAVR Boards" utilizan Timer0 para generar `millis()`. Los núcleos "Placas Arduino ARM (32 bits)" y "Placas Arduino SAMD (32 bits ARM Cortex-M0+)" utilizan el temporizador SysTick.

2.6. Matemáticas

2.6.1. abs()

Descripción:

Calcula el valor absoluto de un número

Sintaxis:

```
abs(x)
```

Parámetros:

x: el número

Devuelve:

x: si x es mayor o igual a 0.
-x: si x es menor que 0.

Código de ejemplo:

Imprime el valor absoluto de la variable x en el monitor serie.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ; // espera a que se conecte el puerto serie. Necesario solo para
     // puerto USB nativo
  }
  int x = 42;
  Serial.print("El valor absoluto de ");
  Serial.print(x);
  Serial.print(" es ");
  Serial.println(abs(x));
  x = -42;
  Serial.print("El valor absoluto de ");
```

```
  Serial.print(x);
  Serial.print(" es ");
  Serial.println(abs(x));
}

void loop() {
}
```

Notas y Advertencias:

Debido a la forma en que se implementa la función abs(), evite usar otras funciones dentro de los corchetes, ya que puede generar resultados incorrectos.

```
abs(a++); // evitar esto - produce resultados incorrectos
// usa esto en su lugar:
abs(a);
a++; // mantener otras matemáticas fuera de la función
```

2.6.2. constrain()

Descripción:

Restringe un número para que esté dentro de un rango.

Sintaxis:

```
constrain(x, a, b)
```

Parámetros:

x: el número a restringir Tipos de datos permitidos: todos los tipos de datos.
a: el extremo inferior del rango. Tipos de datos permitidos: todos los tipos de datos.
b: el extremo superior del rango. Tipos de datos permitidos: todos los tipos de datos.

Devuelve:

x: si x está entre a y b.
a: si x es menor que a.
b: si x es mayor que b.

Código de ejemplo:

El código limita los valores del sensor entre 10 y 150.

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

Notas y advertencias:

Debido a la forma en que se implementa la función de `constrain()`, evite usar otras funciones dentro de los corchetes, ya que puede generar resultados incorrectos.

Este código arrojará resultados incorrectos:

```
int constrainedInput = constrain(Serial.parseInt(), minimumValue,
```

```
maximumValue); // evita esto
```

Usa esto en su lugar:

```
int input = Serial.parseInt(); // mantener otras operaciones fuera de
//la función de restricción
int constrainedInput = constrain(input, minimumValue, maximumValue);
```

2.6.3. map()

Descripción:

Vuelve a asignar un número de un rango a otro. Es decir, un valor **de Bajo/ fromLow** se asignaría **a Bajo/toLow**, un valor **de Alto/fromHigh** se asignará **a Alto/toHigh**, valores intermedios a valores intermedios, etc.

No restringe los valores dentro del rango, porque los valores fuera del rango a veces son intencionados y útiles. La función `constrain()` puede usarse antes o después de esta función, si se desean límites a los rangos.

Tenga en cuenta que los "límites inferiores" de cualquiera de los rangos pueden ser más grandes o más pequeños que los "límites superiores", por lo que la función `map()` se puede usar para invertir un rango de números, por ejemplo:

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

La función también maneja bien los números negativos, por lo que este ejemplo:

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

También es válido y funciona bien.

La función `map()` usa matemáticas enteras, por lo que no generará fracciones, cuando las matemáticas podrían indicar que debería hacerlo. Los restos fraccionarios se truncan y no se redondean ni se promedian.

Sintaxis:

```
map(value, fromLow, fromHigh, toLow, toHigh)
```

Parámetros:

`value`: el número a mapear.
`fromLow`: el límite inferior del rango actual del valor.
`fromHigh`: el límite superior del rango actual del valor.
`toLow`: el límite inferior del rango objetivo del valor.
`toHigh`: el límite superior del rango objetivo del valor.

Devuelve:

El valor mapeado. Tipo de datos: `long`.

Código de ejemplo:

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

Apéndice:

Para los inclinados a las matemáticas, aquí está la función completa:

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

Notas y advertencias:

Como se mencionó anteriormente, la función `map()` usa matemáticas enteras. Entonces, las fracciones pueden suprimirse debido a esto. Por ejemplo, fracciones como 3/2, 4/3, 5/4 se devolverán todas como 1 desde la función `map()`, a pesar de sus diferentes valores reales. Entonces, si su proyecto requiere cálculos precisos (por ejemplo, voltaje con precisión de 3 decimales), considere evitar `map()` e implementar los cálculos manualmente en su código usted mismo.

2.6.4. max()

Descripción:

Calcula el máximo de dos números.

Sintaxis:

```
max(x, y)
```

Parámetros:

x: el primer numero Tipos de datos permitidos: cualquier tipo de datos.
y: el segundo numero Tipos de datos permitidos: cualquier tipo de datos.

Devuelve:

El mayor de los dos valores de parámetro.

Código de ejemplo:

El código asegura que sensVal sea al menos 20.

```
sensVal = max(sensVal, 20); // Asigna sensVal al mayor de sensVal o 20
// (asegurando efectivamente que es al menos 20)
```

Notas y Advertencias:

Tal vez en contra de la intuición, `max()` se usa a menudo para restringir el extremo inferior del rango de una variable, mientras que `min()` se usa para restringir el extremo superior del rango.

Debido a la forma en que se implementa la función `max()`, evite usar otras funciones dentro de los corchetes, ya que puede generar resultados incorrectos.

```
max(a--, 0); // evitar esto - produce resultados incorrectos
// usa esto en su lugar:
max(a, 0);
a--; // mantener otras matemáticas fuera de la función
```

2.6.5. min()

Descripción:

Calcula el mínimo de dos números.

Sintaxis:

```
min(x, y)
```

Parámetros:

x: el primer numero Tipos de datos permitidos: cualquier tipo de datos.
y: el segundo numero Tipos de datos permitidos: cualquier tipo de datos.

Devuelve:

El más pequeño de los dos números.

Código de ejemplo:

El código asegura que nunca supere 100.

```
sensVal = min(sensVal, 100); // asigna sensVal al menor de sensVal o
// 100 asegurándose de que nunca supere los 100
```

Notas y advertencias:

Tal vez en contra de la intuición, `max()` se usa a menudo para restringir el extremo inferior del rango de una variable, mientras que `min()` se usa para restringir el extremo superior del rango.

Debido a la forma en que se implementa la función `min()`, evite usar otras funciones dentro de los corchetes, ya que puede generar resultados incorrectos.

```
min(a++, 100); // evitar esto - produce resultados incorrectos
min(a, 100);
a++; // use esto en su lugar: mantenga otras matemáticas fuera de la
//función
```

2.6.6. pow()

Descripción:

Calcula el valor de un número elevado a una potencia. `pow()` se puede usar para elevar un número a una potencia fraccionaria. Esto es útil para generar un mapeo exponencial de valores o curvas.

Sintaxis:

```
pow(base, exponente)
```

Parámetros:

`base`: el número. Tipos de datos permitidos: `float`.
`exponent`: la potencia a la que se eleva la base. Tipos de datos permitidos `float`.

Devuelve:

El resultado de la exponenciación. Tipo de dato: `double`.

Código de ejemplo:

Calcular el valor de x elevado a la potencia de y:

```
z = pow(x, y);
```

Consulte el sketh (`fscale`) para ver un ejemplo más complejo del uso de `pow()`.

2.6.7. sq()

Descripción:

Calcula el cuadrado de un número: el número multiplicado por sí mismo.

Sintaxis:

```
sq(x)
```

Parámetros:

`x`: el número. Tipos de datos permitidos: cualquier tipo de datos.

Devuelve:

El cuadrado del número. Tipo de dato: `double`.

Notas y Advertencias:

Debido a la forma en que se implementa la función `sq()`, evite usar otras funciones dentro de los orchetes, ya que puede generar resultados incorrectos.

Este código arrojará resultados incorrectos:

```
int inputSquared = sq(Serial.parseInt()); // evita esto
```

Usa esto en su lugar:

```
int input = Serial.parseInt();// mantener otras operaciones fuera de
//la función sq
int inputSquared = sq(input);
```

2.6.8. sqrt()

Descripción:

Calcula la raíz cuadrada de un número.

Sintaxis:

```
sqrt(x)
```

Parámetros:

x: el número. Tipos de datos permitidos: cualquier tipo de datos.

Devuelve:

La raíz cuadrada del número. Tipo de datos: `double`.

2.7. Trigonometría

2.7.1. cos()

Descripción:

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

Sintaxis:

```
cos(rad)
```

Parámetros:

`rad`: El ángulo en radianes. Tipos de datos permitidos: `float`.

Devuelve:

El coseno del ángulo. Tipo de dato: `double`.

2.7.2. sin()

Descripción:

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

Sintaxis:

```
sin(rad)
```

Parámetros:

`rad`: El ángulo en radianes. Tipos de datos permitidos: `float`.

Devuelve:

El seno del ángulo. Tipo de dato: `double`.

2.7.3. tan()

Descripción:

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

Sintaxis:

```
tan(rad)
```

Parámetros:

`rad`: El ángulo en radianes. Tipos de datos permitidos: `float`.

Devuelve:

La tangente del ángulo. Tipo de dato: `double`.

2.8. Caracteres

2.8.1. isAlpha()

Descripción:

Analiza si un char es alfa (eso es una letra). Devuelve verdadero si thisChar contiene una letra.

Sintaxis:

```
isAlpha(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si `thisCrar` es alfa.

Código de ejemplo:

```
if (isAlpha(myChar)) { // comprueba si myChar es una letra
  Serial.println("The character is a letter");
}
else {
  Serial.println("The character is not a letter");
}
```

2.8.2. isAlphaNumeric()

Descripción:

Analiza si un char es alfanumérico (es decir, una letra o un número). Devuelve verdadero si thisChar contiene un número o una letra.

Sintaxis:

```
isAlphaNumeric(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es alfanumérico.

Código de ejemplo:

```
if (isAlphaNumeric(myChar)) { // comprueba si myChar es una letra o un
// número
  Serial.println("The character is alphanumeric");
}
else {
  Serial.println("The character is not alphanumeric");
}
```

2.8.3. isAscii()

Descripción:

Analizar si un char es Ascii. Devuelve verdadero si thisChar contiene un carácter ASCII.

Sintaxis:

```
isAscii(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es Ascii.

Código de ejemplo:

```
if (isAscii(myChar)) { // comprueba si myChar es un carácter ASCII
  Serial.println("The character is Ascii");
}
else {
  Serial.println("The character is not Ascii");
}
```

2.8.4. isControl()

Descripción:

Analizar si un char es un carácter de control. Devuelve verdadero si thisChar es un carácter de control.

Sintaxis:

```
isControl(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es un carácter de control.

Código de ejemplo:

```
if (isControl(myChar)){//comprueba si myChar es un carácter de control
  Serial.println("El caracter es un caracter de control");
}
else {
  Serial.println("El caracter no es un caracter de control");
}
```

2.8.5. isDigit()

Descripción:

Analiza si un char es un dígito (eso es un número). Devuelve verdadero si thisChar es un número.

Sintaxis:

```
isDigit(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es un número.

Código de ejemplo:

```
if (isDigit(myChar)) { // comprueba si myChar es un dígito
  Serial.println("El caracter es un numero");
}
else {
  Serial.println("El carácter no es un numero");
}
```

2.8.6. isGraph()

Descripción:

Analice si un char es imprimible con algún contenido (el espacio es imprimible pero no tiene contenido). Devuelve verdadero si thisChar es imprimible.

Sintaxis:

```
isGraph(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es imprimible.

Código de ejemplo:

```
if (isGraph(myChar)) { // comprueba si myChar es un carácter
  //imprimible pero no un espacio en blanco.
  Serial.println("El carácter es imprimible");
}
else {
  Serial.println("El carácter no es imprimible");
}
```

2.8.7. isHexadecimalDigit()

Descripción:

Analiza si un carácter es un dígito hexadecimal (A-F, 0-9). Devuelve verdadero si thisChar contiene un dígito hexadecimal.

Sintaxis:

```
isHexadecimalDigit(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es un dígito hexadecimal.

Código de ejemplo:

```
if (isHexadecimalDigit(myChar)) { // comprueba si myChar es un dígito
  //hexadecimal
  Serial.println("El carácter es un dígito hexadecimal.");
}
else {
  Serial.println("El carácter no es un dígito hexadecimal.");
}
```

2.8.8. isLowerCase()

Descripción:

Analiza si un char es minúscula (es decir, una letra en minúscula). Devuelve verdadero si thisChar contiene una letra en minúsculas.

Sintaxis:

```
isLowerCase(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar está en minúsculas.

Código de ejemplo:

```
if (isLowerCase(myChar)) { // comprueba si myChar es una letra
  Serial.println("El carácter está en minúsculas");
}
else {
  Serial.println("El carácter no está en minúsculas");
}
```

2.8.9. isUpperCase()

Descripción:

Analiza si un char está en mayúsculas (es decir, una letra en mayúsculas). Devuelve verdadero si thisChar está en mayúsculas

Sintaxis:

```
isUpperCase(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar está en mayúsculas.

Código de ejemplo:

```
if (isUpperCase(myChar)) { // prueba si myChar es una letra mayúscula
  Serial.println("El caracter esta en mayusculas");
}
else {
  Serial.println("El carácter no está en mayúsculas.");
}
```

2.8.10. isPrintable()

Descripción:

Analiza si un carácter es imprimible (es decir, cualquier carácter que produzca una salida, incluso un espacio en blanco). Devuelve verdadero si thisChar es imprimible.

Sintaxis:

```
isPrintable(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es imprimible.

Código de ejemplo:

```
if (isPrintable(myChar)) { // comprueba si myChar es un carácter
 // imprimible
  Serial.println("El carácter es imprimible");
}
else {
  Serial.println("El carácter no es imprimible");
}
```

2.8.11. isPunct()

Descripción:

Analiza si un carácter es puntuación (es decir, una coma, un punto y coma, un signo de exclamación, etc.). Devuelve verdadero si thisChar es puntuación.

Sintaxis:

```
isPunct(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: verdadero: si thisChar es un signo de puntuación.

Código de ejemplo:

```
if (isPunct(myChar)) { // prueba si myChar es un carácter de puntuación
  Serial.println("El carácter es una puntuación.");
}
else {
  Serial.println("El carácter no es una puntuación.");
}
```

2.8.11. isSpace()

Descripción:

Analice si un carácter es un carácter de espacio en blanco. Devuelve verdadero si el argumento es un espacio, avance de página (`'\f'`), nueva línea (`'\n'`), retorno de carro (`'\r'`), tabulación horizontal (`'t'`) o tabulación vertical (`'\v'`).

Sintaxis:

```
isSpace(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: verdadero: si thisChar es un carácter de espacio en blanco.

Código de ejemplo:

```
if (isSpace(myChar)) { // prueba si myChar es un carácter de
  //espacio en
  Serial.println("El caracter es un espacio en blanco");
}
else {
  Serial.println("El caracter no es un espacio en blanco");
}
```

2.8.13. isWhitespace()

Descripción:

Analiza si un carácter es un carácter de espacio. Devuelve verdadero si el argumento es un espacio o una tabulación horizontal (`'\t'`).

Sintaxis:

```
isWhitespace(thisChar)
```

Parámetros:

`thisChar`: variable. Tipos de datos permitidos: `char`.

Devuelve:

`true`: si thisChar es un carácter de espacio.

Código de ejemplo:

```
if (isWhitespace(myChar)) { // prueba si myChar es un carácter de espacio
  Serial.println("El caracter es un espacio o tabulado");
}
else {
  Serial.println("El caracter no es un espacio o tabulado");
}
```

2.9. Random Numbers

2.9.1. random()

Descripción:

La función aleatoria genera números pseudoaleatorios.

Sintaxis:

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

Parámetros:

`min`: límite inferior del valor aleatorio, inclusive (opcional).
`max`: límite superior del valor aleatorio, exclusivo.

Devuelve:

Un número aleatorio entre min y max-1. Tipo de dato: `long`.

Código de ejemplo:

El código genera números aleatorios y los muestra.

```
long randNumber;

void setup() {
  Serial.begin(9600);
  // si el pin de entrada analógica 0 no está conectado, analógico
  // aleatorio
  // el ruido hará que la llamada a randomSeed() genere diferentes
  // números de inicialización cada vez que se ejecuta el boceto.
  // randomSeed() luego barajará la función aleatoria.
  randomSeed(analogRead(0));
}
void loop() {
  // imprimir un número aleatorio de 0 a 299
  randNumber = random(300);
  Serial.println(randNumber);
  // imprimir un número aleatorio del 10 al 19
  randNumber = random(10, 20);
  Serial.println(randNumber);
  delay(50);
}
```

Notas y Advertencias:

Si es importante que una secuencia de valores generados por `random()` difiera, en ejecuciones posteriores de un boceto, use `randomSeed()` para inicializar el generador de números aleatorios con una entrada bastante aleatoria, como `analogRead()` en un pin desconectado.

Por el contrario, ocasionalmente puede ser útil usar secuencias pseudoaleatorias que se repiten exactamente. Esto se puede lograr llamando a `randomSeed()` con un número fijo, antes de comenzar la

secuencia aleatoria.

El parámetro `max` debe elegirse de acuerdo con el tipo de datos de la variable en la que se almacena el valor. En cualquier caso, el máximo absoluto está ligado a la naturaleza `long` del valor generado (32 bit - 2.147.483.647). Establecer `max` en un valor más alto no generará un error durante la compilación, pero durante la ejecución del boceto, los números generados no serán los esperados.

2.9.2. randomSeed()

Descripción:

`randomSeed()` inicializa el generador de números pseudoaleatorios, lo que hace que comience en un punto arbitrario de su secuencia aleatoria. Esta secuencia, aunque muy larga y aleatoria, es siempre la misma.

Si es importante que una secuencia de valores generados por `random()` difiera, en ejecuciones posteriores de un boceto, use `randomSeed()` para inicializar el generador de números aleatorios con una entrada bastante aleatoria, como `analogRead()` en un pin desconectado.

Por el contrario, ocasionalmente puede ser útil usar secuencias pseudoaleatorias que se repiten exactamente. Esto se puede lograr llamando a `randomSeed()` con un número fijo, antes de comenzar la secuencia aleatoria.

Sintaxis:

```
randomSeed(seed)
```

Parámetros:

seed: número para inicializar la secuencia pseudoaleatoria. Tipos de datos permitidos: `unsigned long`.

Devuelve:

Nada.

Código de ejemplo:

El código genera un número pseudoaleatorio y envía el número generado al puerto serie.

```
long randNumber;

void setup() {
  Serial.begin(9600);
  randomSeed(analogRead(0));
}
void loop() {
  randNumber = random(300);
  Serial.println(randNumber);
  delay(50);
}
```

2.10. Bits and Bytes

2.10.1. bit()

Descripción:

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

Sintaxis:

```
bit(n)
```

Parámetros:

n: el bit cuyo valor a calcular

Devuelve:

El valor del bit.

2.10.2. bitClear()

Descripción:

Borra (escribe un 0 en) un bit de una variable numérica.

Sintaxis:

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

Parámetros:

x: la variable numérica cuyo bit se va a borrar.
n: qué bit borrar, comenzando en 0 para el bit menos significativo (más a la derecha).

Devuelve:

x: el valor de la variable numérica después de que se borra el bit en la posición `n`.

Código de ejemplo:

Imprime la salida de `bitClear(x,n)` en dos enteros dados. La representación binaria de 6 es 0110, por lo que cuando `n=1`, el segundo bit de la derecha se establece en 0. Después de esto, nos queda 0100 en binario, por lo que se devuelve 4.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ; // espere a que se conecte el puerto serie. Necesario solo para
      // puerto USB nativo
  }

  int x = 6;
  int n = 1;
```

```
  Serial.print(bitClear(x, n)); // imprime la salida de bitClear(x,n)
}

void loop() {
}
```

2.10.3. bitRead()

Descripción:

Lee un bit de un número.

Sintaxis:

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

Parámetros:

x: El número desde el que leer.
n: qué bit leer, comenzando en 0 para el bit menos significativo (más a la derecha).

Devuelve:

El valor del bit (0 o 1).

2.10.4. bitSet()

Descripción:

Establece (escribe un 1 en) un bit de una variable numérica.

Sintaxis:

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

Parámetros:

x: la variable numérica cuyo bit se va a establecer.
n: qué bit configurar, comenzando en 0 para el bit menos significativo (más a la derecha).

Devuelve:

Nada.

2.10.5. bitWrite()

Descripción:

Escribe un poco de una variable numérica.

Sintaxis:

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

Parámetros:

x: la variable numérica en la que escribir.
n: qué bit del número escribir, comenzando en 0 para el bit menos significativo (más a la derecha).
b: el valor a escribir en el bit (0 o 1).

Devuelve:

Nada.

Código de ejemplo:

Demuestra el uso de bitWrite imprimiendo el valor de una variable en el monitor serie antes y después del uso de `bitWrite()`.

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {} // espere a que se conecte el puerto serie.
  // Necesario solo para puerto USB nativo
  byte x = 0b10000000; // el prefijo 0b indica una constante binaria
  Serial.println(x, BIN); // 10000000
  bitWrite(x, 0, 1); // escribir 1 al bit menos significativo de x
  Serial.println(x, BIN); // 10000001
}
void loop() {}
```

2.10.6. highByte()

Descripción:

Extrae el byte de orden superior (más a la izquierda) de una palabra (o el segundo byte más bajo de un tipo de datos más grande).

Sintaxis:

```
highByte(x)
```

Parámetros:

x: un valor de cualquier tipo

Devuelve:

Tipo de dato: `byte`.

2.10.7. lowByte()

Descripción:

Extrae el byte de orden inferior (más a la derecha) de una variable (ej., una palabra).

Sintaxis:

```
lowByte(x)
```

Parámetros:

`x`: un valor de cualquier tipo.

Devuelve:

Tipo de dato: `byte`.

2.11. Interrupciones

2.11.1. interrupts()

Descripción:

Vuelve a habilitar las interrupciones (después de que `noInterrupts()` las haya deshabilitado). Las interrupciones permiten que se realicen ciertas tareas importantes en segundo plano y están habilitadas de manera predeterminada. Algunas funciones no funcionarán mientras las interrupciones estén deshabilitadas, y la comunicación entrante puede ignorarse. Interrupciones Sin embargo, puede interrumpir levemente la sincronización del código y puede desactivarse para secciones de código particularmente críticas.

Sintaxis:

```
interrupts()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

Código de ejemplo:

El código muestra cómo habilitar las interrupciones.

```
void setup() {}

void loop() {
  noInterrupts();
  // código crítico, sensible al tiempo aquí
  interrupts();
```

```
  // other code here
}
```

2.11.2. noInterrupts()

Descripción:

Deshabilita las interrupciones (puede volver a habilitarlas con `interrupts()`). Las interrupciones permiten que ciertas tareas importantes sucedan en segundo plano y están habilitadas de forma predeterminada. Algunas funciones no funcionarán mientras las interrupciones estén deshabilitadas y la comunicación entrante puede ignorarse. Sin embargo, las interrupciones pueden alterar levemente la sincronización del código y pueden desactivarse para secciones de código particularmente críticas.

Sintaxis:

```
noInterrupts()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

Código de ejemplo:

El código muestra cómo habilitar las interrupciones.

```
void setup() {}

void loop() {
  noInterrupts();
  // código crítico, sensible al tiempo aquí
  interrupts();
  // otro código aquí
}
```

Tenga en cuenta que deshabilitar las interrupciones en las placas Arduino con capacidades USB nativas (por ejemplo, Leonardo) hará que la placa no aparezca en el menú Puerto, ya que esto deshabilita su capacidad USB.

2.12. Interrupciones externas

2.12.1. attachInterrupt()

Descripción:

Pines digitales con interrupciones

El primer parámetro de `attachInterrupt()` es un número de interrupción. Normalmente, debe usar `digitalPinToInterrupt(pin)` para traducir el pin digital real al número de interrupción específico. Por ejemplo, si se conecta al pin 3, use `digitalPinToInterrupt(3)` como primer parámetro para

`addedInterrupt()`.

PLACA	PINES DIGITALES UTILIZABLES PARA INTERRUPCIONES
Uno, Nano, Mini, other 328-based	2, 3
Uno WiFi Rev.2, Nano Every	Todos los pines digitales
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21 (los **pines 20 y 21** no están disponibles para interrupciones mientras se usan para comunicación I2C)
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	Todos los pines digitales, excepto el 4
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Nano 33 IoT	2, 3, 9, 10, 11, 13, A1, A5, A7
Nano 33 BLE, Nano 33 BLE Sense	Todos los pines
Due	Todos los pines digitales
101	todos los pines digitales (solo los pines 2, 5, 7, 8, 10, 11, 12, 13 funcionan con **CAMBIO**)

Notas y advertencias:

Nota

Dentro de la función adjunta, `delay()` no funcionará y el valor devuelto por `millis()`) no se incrementará. Los datos en serie recibidos durante la función pueden perderse. Debe declarar como `volatile` cualquier variable que modifique dentro de la función adjunta. Consulte la sección sobre ISR a continuación para obtener más información.

Uso de interrupciones

Las interrupciones son útiles para hacer que las cosas sucedan automáticamente en los programas de microcontroladores y pueden ayudar a resolver problemas de temporización. Las buenas tareas para usar una interrupción pueden incluir leer un codificador rotatorio o monitorear la entrada del usuario.

Si quisiera asegurarse de que un programa siempre captara los pulsos de un codificador rotatorio, para que nunca pierda un pulso, sería muy complicado escribir un programa para hacer cualquier otra cosa, porque el programa necesitaría sondear constantemente el sensor. líneas para el codificador, con el fin de capturar pulsos cuando se produjeron. Otros sensores también tienen una dinámica de interfaz similar, como intentar leer un sensor de sonido que intenta captar un clic, o un sensor de ranura de infrarrojos (fotointerruptor) que intenta captar la caída de una moneda. En todas estas situaciones, el uso de una interrupción puede liberar al microcontrolador para realizar otro trabajo sin perder la entrada.

Acerca de las rutinas de servicio de interrupción

Los ISR son tipos especiales de funciones que tienen algunas limitaciones únicas que la mayoría de las otras funciones no tienen. Un ISR no puede tener ningún parámetro y no debería devolver nada.

En general, una ISR debe ser lo más corta y rápida posible. Si su boceto usa múltiples ISR, solo se puede ejecutar uno a la vez, otras interrupciones se ejecutarán después de que finalice la actual en un orden que

depende de la prioridad que tengan. `millis()` se basa en las interrupciones para contar, por lo que nunca se incrementará dentro de un ISR. Dado que `delay()` requiere interrupciones para funcionar, no funcionará si se llama dentro de un ISR. `micros()` funciona inicialmente pero comenzará a comportarse de manera errática después de 1-2 ms. `delayMicroseconds()` no utiliza ningún contador, por lo que funcionará con normalidad.

Normalmente, las variables globales se utilizan para pasar datos entre un ISR y el programa principal. Para asegurarse de que las variables compartidas entre un ISR y el programa principal se actualicen correctamente, declárelas como volátiles.

Para obtener más información sobre las interrupciones, consulte las notas de Nick Gammon. (http://gammon.com.au/interrupts)

Sintaxis:

`attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)` (recomendado)
`attachInterrupt(interrupt, ISR, mode)` (no recomendado)
`attachInterrupt(pin, ISR, mode)` (no recomendado. Además, esta sintaxis solo funciona en Arduino SAMD Boards, Uno WiFi Rev2, Due, and 101.)

Parámetros:

`interrupt`: el número de la interrupción. Tipos de datos permitidos: `int`.
`pin` : el número de pin de Arduino.
`ISR` : el ISR a llamar cuando ocurre la interrupción; esta función no debe tomar parámetros y no devolver nada. Esta función a veces se denomina rutina de servicio de interrupción.
`mode` : define cuándo se debe disparar la interrupción. Cuatro constantes están predefinidas como valores válidos:

- **LOW** para activar la interrupción siempre que el pin esté bajo,
- **CHANGE** para activar la interrupción cada vez que el pin cambie de valor
- **RISING** para disparar cuando el pin va de bajo a alto,
- **FALLING** para cuando el pin va de mayor a menor.

Las tarjetas Due, Zero y MKR1000 también permiten:

- **HIGH** para activar la interrupción siempre que el pin esté alto.

Devuelve:

Nada.

Código de ejemplo:

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;
void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}
void loop() {
  digitalWrite(ledPin, state);
}
void blink() {
  state = !state;
}
```

Números de interrupción

Normalmente, debe usar `digitalPinToInterrupt(pin)` en lugar de colocar un número de interrupción directamente en su boceto. Los pines específicos con interrupciones y su asignación al número de interrupciones varían para cada tipo de placa. El uso directo de números de interrupción puede parecer simple, pero puede causar problemas de compatibilidad cuando su boceto se ejecuta en un tablero diferente.

Sin embargo, los bocetos más antiguos a menudo tienen números de interrupción directos. A menudo se usaba el número 0 (para el pin digital 2) o el número 1 (para el pin digital 3). La siguiente tabla muestra los pines de interrupción disponibles en varias placas.

Tenga en cuenta que en la siguiente tabla, los números de interrupción se refieren al número que se pasará a `attachInterrupt()`. Por razones históricas, esta numeración no siempre corresponde directamente a la numeración de interrupciones en el chip ATmega (por ejemplo, int.0 corresponde a INT4 en el chip ATmega2560).

BOARD	INT.0	INT.1	INT.2	INT.3	INT.4	INT.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	

Para las placas Uno WiFiRev.2, Due, Zero, MKR Family y 101, el **número de interrupción = número de pin**.

2.12.2. detachInterrupt()

Descripción:

Desactiva la interrupción dada.

Sintaxis:

`detachInterrupt(digitalPinToInterrupt(pin))` (recomendado)
`detachInterrupt(interrupt)` (no recomendado)
`detachInterrupt(pin)` (No recomendado. Además, esta sintaxis solo funciona en Arduino SAMD Boards, Uno WiFi Rev2, Due, and 101.)

Parámetros:

`interrupt`: el número de la interrupción a deshabilitar (vea `attachInterrupt()` para más detalles).
`pin`: el número de pin de Arduino de la interrupción para deshabilitar.

Devuelve:

Nada.

2.13. Comunicaciones

2.13.1. Serial

Descripción:

Se utiliza para la comunicación entre la placa Arduino y una computadora u otros dispositivos. Todas las placas Arduino tienen al menos un puerto serie (también conocido como UART o USART), y algunas tienen varios.

BOARD	USB CDC NAME	SERIAL PINS	SERIAL1 PINS	SERIAL2 PINS	SERIAL3 PINS
Uno, Nano, Mini		0(RX), 1(TX)			
Mega		0(RX), 1(TX)	19(RX), 18(TX)	17(RX), 16(TX)	15(RX), 14(TX)
Leonardo, Micro, Yún	Serial		0(RX), 1(TX)		
Uno WiFi Rev.2		Connected to USB	0(RX), 1(TX)	Connected to NINA	
MKR boards	Serial		13(RX), 14(TX)		
Zero	SerialUSB (Native USB Port only)	Connected to Programming Port	0(RX), 1(TX)		
Due	SerialUSB (Native USB Port only)	0(RX), 1(TX)	19(RX), 18(TX)	17(RX), 16(TX)	15(RX), 14(TX)
101	Serial		0(RX), 1(TX)		

En Uno, Nano, Mini y Mega, los pines 0 y 1 se utilizan para la comunicación con la computadora. Conectar cualquier cosa a estos pines puede interferir con esa comunicación, incluso causar cargas fallidas a la placa.

Puede utilizar el monitor serie integrado del entorno Arduino para comunicarse con una placa Arduino. Haga clic en el botón del monitor serie en la barra de herramientas y seleccione la misma velocidad en baudios utilizada en la llamada a `begin()`.

La comunicación serial en los pines TX/RX usa niveles lógicos TTL (5V o 3.3V dependiendo de la placa). No conecte estos pines directamente a un puerto serial RS232; funcionan a +/- 12V y pueden dañar su placa Arduino.

Para usar estos puertos serie adicionales para comunicarse con su computadora personal, necesitará un adaptador USB a serie adicional, ya que no están conectados al adaptador USB a serie del Mega. Para usarlos para comunicarse con un dispositivo serie TTL externo, conecte el pin TX al pin RX de su dispositivo, el RX al pin TX de su dispositivo y la tierra de su Mega a la tierra de su dispositivo.

2.13.1.1. if(Serial)

Descripción:

Indica si el puerto serie especificado está listo.

En las placas con USB nativo, `if(Serial)` (o `if(SerialUSB)` en Due) indica si la conexión serial USB CDC está abierta o no. Para todas las demás placas y los puertos CDC que no son USB, esto siempre será verdadero.

Esto se introdujo en Arduino IDE 1.0.1.

Sintaxis:

```
if(Serial)
```

Parámetros:

Ninguno.

Devuelve:

Devuelve verdadero si el puerto serie especificado está disponible. Esto solo devolverá falso si consulta la conexión en serie USB CDC de Leonardo antes de que esté lista. Tipo de dato: `bool`.

Código de ejemplo:

```
void setup() {
  //Inicialice la serie y espere a que se abra el puerto:
  Serial.begin(9600);
  while (!Serial) {
    ; // espere a que se conecte el puerto serie. Necesario para USB
      // nativo
  }
}
void loop() {
  //proceda normalmente
}
```

2.13.1.2. Serial.available()

Descripción:

Obtenga la cantidad de bytes (caracteres) disponibles para leer desde el puerto serie. Estos son datos que ya llegaron y se almacenaron en el búfer de recepción en serie (que contiene 64 bytes).

`available()` hereda de la clase Stream.

Sintaxis:

```
Serial.available()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal Serie (2.13.1.).

Devuelve:

El número de bytes disponibles para leer.

Código de ejemplo:

El siguiente código devuelve un carácter recibido a través del puerto serie.

```
int incomingByte = 0; // para datos en serie entrantes

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

void loop() {
  // responde solo cuando recibas datos:
  if (Serial.available() > 0) {
    // leer el byte entrante:
    incomingByte = Serial.read();
    // di lo que tienes:
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

Arduino Mega ejemplo:

Este código envía los datos recibidos en un puerto serie del Arduino Mega a otro. Esto se puede usar, por ejemplo, para conectar un dispositivo serie a la computadora a través de la placa Arduino.

```
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
}
void loop() {
  // read from port 0, send to port 1:
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, DEC);
  }
  // read from port 1, send to port 0:
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, DEC);
  }
}
```

2.13.1.3. Serial.availableForWire()

Descripción:

Obtenga la cantidad de bytes (caracteres) disponibles para escribir en el búfer en serie sin bloquear la

operación de escritura.

Sintaxis:

```
Serial.availableForWrite()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal Serie (2.13.1.).

Devuelve:

El número de bytes disponibles para escribir.

2.13.1.4. Serial.begin()

Descripción:

Establece la velocidad de datos en bits por segundo (baudios) para la transmisión de datos en serie. Para comunicarse con Serial Monitor, asegúrese de usar una de las velocidades en baudios enumeradas en el menú en la esquina inferior derecha de su pantalla. Sin embargo, puede especificar otras velocidades, por ejemplo, para comunicarse a través de los pines 0 y 1 con un componente que requiere una velocidad en baudios particular.

Un segundo argumento opcional configura los bits de datos, paridad y parada. El valor predeterminado es 8 bits de datos, sin paridad, un bit de parada.

Sintaxis:

```
Serial.begin(speed)
Serial.begin(speed, config)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie. (2.13.1)
`speed`: en bits por segundo (baudios). Tipos de datos permitidos: `long`.
`config`: establece datos, paridad y bits de parada. Los valores válidos son:

- `SERIAL_5N1`
- `SERIAL_6N1`
- `SERIAL_7N1`
- `SERIAL_8N1` (el valor por defecto)
- `SERIAL_5N2`
- `SERIAL_6N2`
- `SERIAL_7N2`
- `SERIAL_8N2`
- `SERIAL_5E1`: incluso la paridad
- `SERIAL_6E1`
- `SERIAL_7E1`
- `SERIAL_8E1`
- `SERIAL_5E2`
- `SERIAL_6E2`
- `SERIAL_7E2`

SERIAL_8E2
SERIAL_5O1: paridad impar
SERIAL_6O1
SERIAL_7O1
SERIAL_8O1
SERIAL_5O2
SERIAL_6O2
SERIAL_7O2
SERIAL_8O2

Devuelve:

Nada

Código de ejemplo:

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

Ejemplo Arduino Mega:

```
// Arduino Mega usando sus cuatro puertos serie
// (Serial, Serial1, Serial2, Serial3)
// con diferentes velocidades de transmisión:
void setup() {
  Serial.begin(9600);
  Serial1.begin(38400);
  Serial2.begin(19200);
  Serial3.begin(4800);

  Serial.println("Hello Computer");
  Serial1.println("Hello Serial 1");
  Serial2.println("Hello Serial 2");
  Serial3.println("Hello Serial 3");
}
void loop() {}
```

Notas y advertencias:

Para los puertos seriales USB CDC (por ejemplo, `Serial` en Leonardo), `Serial.begin()` es irrelevante. Puede utilizar cualquier velocidad de transmisión y configuración para la comunicación en serie con estos puertos. Consulte la lista de puertos serie disponibles para cada placa en la página principal Serial.

El único valor de `config` compatible con Serial1 en las placas Arduino Nano 33 BLE y Nano 33 BLE Sense es `SERIAL_8N1`.

2.13.1.5. Serial.end()

Descripción:

Deshabilita la comunicación en serie, lo que permite que los pines RX y TX se usen para entrada y salida general. Para volver a habilitar la comunicación en serie, llame a `Serial.begin()`.

Sintaxis:

```
Serial.end()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.

Devuelve:

Nada.

2.13.1.6. Serial.find()

Descripción:

`Serial.find()` lee datos del búfer serial hasta que se encuentra el objetivo. La función devuelve `true` si se encuentra el objetivo, `false` si se agota el tiempo de espera.

`Serial.find()` hereda de la clase de utilidad de stream

Sintaxis:

```
Serial.find(target)
Serial.find(target, length)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de Serial.
`target`: la cadena a buscar. Tipos de datos permitidos: `char`.
`length`: longitud del objetivo. Tipos de datos permitidos: `size_t`.

Devuelve:

Tipo de datos: `bool`.

2.13.1.7. Serial.findUntil()

Descripción:

`Serial.findUntil()` lee datos del búfer serial hasta que se encuentra una cadena de destino de longitud dada o una cadena de terminación.

La función devuelve verdadero si se encuentra la cadena de destino, falso si se agota el tiempo de espera.

`Serial.findUntil()` hereda de la clase de Stream

Sintaxis:

```
Serial.findUntil(target, terminal)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.
`target`: la cadena a buscar. Tipos de datos permitidos: `char`.
`terminal`: la cadena terminal en la búsqueda. Tipos de datos permitidos: `char`.

Devuelve:

Tipo de dato: `bool`.

2.13.1.8. Serial.flush()

Descripción:

Espera a que se complete la transmisión de los datos seriales salientes. (Antes de Arduino 1.0, esto eliminaba los datos en serie entrantes almacenados en el búfer).

`flush()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.flush()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.

Devuelve:

Nada.

2.13.1.9. Serial.parseFloat()

Descripción:

`Serial.parseFloat()` devuelve el primer número de punto flotante válido del búfer Serial. `parseFloat()` termina con el primer carácter que no es un número de coma flotante. La función termina si se agota el tiempo de espera (ver `Serial.setTimeout()`).

`Serial.parseFloat()` hereda de la clase de utilidad `Stream`.

Sintaxis:

```
Serial.parseFloat()
Serial.parseFloat(lookahead)
Serial.parseFloat(lookahead, ignore)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de Serial.
`lookahead`: el modo utilizado para mirar hacia adelante en la secuencia en busca de un número de coma flotante. Tipos de datos permitidos: `LookaheadMode`. Valores `lookahead` permitidos:

- `SKIP_ALL`: todos los caracteres que no sean el signo menos, el punto decimal o los dígitos se ignoran al escanear la transmisión en busca de un número de punto flotante. Este es el modo por defecto.
- `SKIP_NONE`: No se salta nada y la secuencia no se toca a menos que el primer carácter de espera sea válido.
- `SKIP_WHITESPACE`: Solo se saltan los tabuladores, los espacios, los saltos de línea y los retornos de carro.

`ignore`: utilizado para omitir el carácter indicado en la búsqueda. Se utiliza, por ejemplo, para omitir el divisor de miles. Tipos de datos permitidos: `char`

Devuelve:

Tipo de datos: `float`.

2.13.1.10. Serial.parseInt()

Descripción:

Busca el siguiente entero válido en la serie entrante. La función finaliza si se agota el tiempo de espera (ver `Serial.setTimeout()`).

`Serial.parseInt()` hereda de la clase de utilidad Stream.

En particular:

- El análisis se detiene cuando no se han leído caracteres para un valor de tiempo de espera configurable, o se lee un dígito que no es;
- Si no se leyeron dígitos válidos cuando se agotó el tiempo de espera (consulte `Serial.setTimeout()`), se devuelve 0.

Sintaxis:

```
Serial.parseInt()
Serial.parseInt(lookahead)
Serial.parseInt(lookahead, ignore)
```

Parámetros:

`Serial` : objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal Serie.
`lookahead` : el modo utilizado para mirar hacia adelante en la secuencia en busca de un número entero. Tipos de datos permitidos: `LookaheadMode` . Valores `lookahead` permitidos:

- `SKIP_ALL`: todos los caracteres que no sean dígitos o un signo menos se ignoran al escanear la secuencia en busca de un número entero. Este es el modo por defecto.

- `SKIP_NONE` : No se salta nada y la transmisión no se toca a menos que el primer carácter de espera sea válido.
- `SKIP_WHITESPACE`: Solo se saltan los tabuladores, los espacios, los saltos de línea y los retornos de carro.

`ignore`: utilizado para omitir el carácter indicado en la búsqueda. Se utiliza, por ejemplo, para omitir el divisor de miles. Tipos de datos permitidos: `char`.

Devuelve:

El siguiente entero válido. Tipo de dato: `long`.

2.13.1.11. Serial.peek()

Descripción:

Devuelve el siguiente byte (carácter) de los datos seriales entrantes sin eliminarlos del búfer serial interno. Es decir, las llamadas sucesivas a `peek()` devolverán el mismo carácter, al igual que la siguiente llamada a `read()`.

`Serial.peek()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.peek()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de Serial.

Devuelve:

El primer byte de datos en serie entrantes disponibles (o -1 si no hay datos disponibles). Tipo de datos: `int`.

2.13.1.12. Serial.print()

Descripción:

Imprime datos en el puerto serial como texto ASCII legible por humanos. Este comando puede tomar muchas formas. Los números se imprimen utilizando un carácter ASCII para cada dígito. Los flotantes se imprimen de manera similar como dígitos ASCII, por defecto con dos decimales. Los bytes se envían como un solo carácter. Los caracteres y las cadenas se envían tal cual. Por ejemplo:

- `Serial.print(78)` da "78"
- `Serial.print(1.23456)` da "1.23"
- `Serial.print('N')` da "N"
- `Serial.print("Hello world.")` da "Hello world."

Un segundo parámetro opcional especifica la base (formato) a usar; los valores permitidos son `BIN (binario o base 2),OCT (octal o base 8),DEC (decimal o base 10),HEX`

`(hexadecimal o base 16)`. Para números de punto flotante, este parámetro especifica el número de lugares decimales que se usarán. Por ejemplo:

- `Serial.print(78, BIN)` da "1001110"
- `Serial.print(78, OCT)` da "76"
- `Serial.print(78, DEC)` da "78"
- `Serial.print(78, HEX)` da "4E"
- `Serial.print(1.23456, 0)` da "1"
- `Serial.print(1.23456, 2)` da "1.23"
- `Serial.print(1.23456, 4)` da " 1.2346"

Puede pasar cadenas basadas en memoria flash a `Serial.print()` envolviéndolas con `F()`. Por ejemplo:

```
Serial.print(F("Hello World"))
```

Para enviar datos sin conversión a su representación como caracteres, use `Serial.write()`.

Sintaxis:

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

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.
`val`: el valor a imprimir. Tipos de datos permitidos: cualquier tipo de datos.

Devuelve:

`print()` devuelve el número de bytes escritos, aunque leer ese número es opcional. Tipo de datos: `size_t`.

Código de ejemplo:

```
/*
Utiliza un bucle for para imprimir números en varios formatos..
*/

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

void loop() {
  // imprimir etiquetas
  Serial.print("SIN FORMATO"); // imprime una etiqueta
  Serial.print("\t"); // imprime una tab
  Serial.print("DEC");
  Serial.print("\t");
  Serial.print("HEX");
  Serial.print("\t");
  Serial.print("OCT");
  Serial.print("\t");
  Serial.print("BIN");
  Serial.println(); // retorno de carro después de la última etiqueta
```

```
  for (int x = 0; x < 64; x++) { // solo una parte de la tabla ASCII,
  // cambie para adaptarla imprimirlo en muchos formatos:
    Serial.print(x); // imprimir como un decimal codificado en ASCII -
    // igual que "DEC"
    Serial.print("\t\t"); // imprime dos pestañas para adaptarse a la
    // longitud de la etiqueta
    Serial.print(x, DEC); // imprimir como un decimal codificado en
    // ASCII
    Serial.print("\t"); // imprime una tab
    Serial.print(x, HEX); // Imprimir como un hexadecimal codificado
    // en ASCII
    Serial.print("\t"); // imprime una tab
    Serial.print(x, OCT); // imprimir como un octal codificado en
    // ASCII
    Serial.print("\t"); // imprime una tab
    Serial.println(x, BIN); // imprimir como un binario codificado en
    // ASCII luego agrega el retorno de carro con "println"
    delay(200); // retraso 200 milisegundos
  }
  Serial.println(); // imprime otro retorno de carro
}
```

Notas y advertencias:

Para obtener información sobre la asincronía de `Serial.print()`, consulte la sección Notas y advertencias de la página de referencia de `Serial.write()`.

2.13.1.13. Serial.println()

Descripción:

Imprime datos en el puerto serie como texto ASCII legible por humanos seguido de un carácter de retorno de carro (ASCII 13 o `'\r'`) y un carácter de nueva línea (ASCII 10 o `'\n'`). Este comando toma las mismas formas que `Serial.print()`.

Sintaxis:

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

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.
`val`: el valor a imprimir. Tipos de datos permitidos: cualquier tipo de datos.
`format`: especifica la base numérica (para tipos de datos integrales) o el número de posiciones decimales (para tipos de punto flotante).

Devuelve:

`println()` devuelve el número de bytes escritos, aunque leer ese número es opcional. Tipo de datos: `size_t`.

Código de ejemplo:

```
// La entrada analógica lee una entrada analógica en la entrada
// analógica  0 imprime el valor.

int analogValue = 0; // variable para mantener el valor analógico

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

void loop() {

  // leer la entrada analógica en el pin 0:
  analogValue = analogRead(0); // imprimirlo en muchos formatos:

  Serial.println(analogValue); // imprimir como un decimal
  // codificado en ASCII
  Serial.println(analogValue, DEC); // imprimir como un decimal
  // codificado en ASCII
  Serial.println(analogValue, HEX); // Imprimir como un hexadecimal
  // codificado en ASCII
  Serial.println(analogValue, OCT); // imprimir como un octal
  // codificado en ASCII
  Serial.println(analogValue, BIN); // imprimir como un binario

  //codificado en ASCII imprimir como un binario codificado en ASCII
  delay(10);
}
```

Notas y advertencias:

Para obtener información sobre la asincronía de `Serial.println()`, consulte la sección Notas y advertencias de la página de referencia de `Serial.write()`.

2.13.1.14. Serial.read()

Descripción:

Lee los datos seriales entrantes.

`Serial.read()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.read()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.

Devuelve:

El primer byte de datos en serie entrantes disponibles (o -1 si no hay datos disponibles). Tipo de dato: `int`.

Código de ejemplo:

```
int incomingByte = 0; // para datos en serie entrantes
void setup() {
  Serial.begin(9600); // abre el puerto serie, establece la velocidad
  // de datos en 9600 bps
}
void loop() {
  // enviar datos solo cuando reciba datos:
  if (Serial.available() > 0) {
    // leer el byte entrante:
    incomingByte = Serial.read();
    // di lo que tienes:
    Serial.print("I received: ");
    Serial.println(incomingByte, DEC);
  }
}
```

2.13.1.15. Serial.readBytes()

Descripción:

`Serial.readBytes()` lee caracteres del puerto serie en un búfer. La función finaliza si se ha leído la longitud determinada o se agota el tiempo de espera (ver `Serial.setTimeout()`).

`Serial.readBytes()` devuelve el número de caracteres colocados en el búfer. Un 0 significa que no se encontraron datos válidos.

`Serial.readBytes()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.readBytes(buffer, length)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal Serial.
`buffer`: El búfer para almacenar los bytes. Tipos de datos permitidos: array de `char` o `byte`.
`length`: el número de bytes a leer. Tipos de datos permitidos: `int`.

Devuelve:

El número de bytes colocados en el búfer. Tipo de datos: `size_t`.

2.13.1.16. Serial.readBytesUntil()

Descripción:

`Serial.readBytesUntil()` lee caracteres del búfer serial en una matriz. La función finaliza (las comprobaciones se realizan en este orden) si se ha leído la longitud determinada, si se agota el tiempo de espera (consulte `Serial.setTimeout()`) o si se detecta el carácter de terminación (en cuyo caso, la función devuelve los caracteres hasta el último carácter antes del terminador suministrado). El terminador en sí no se devuelve en el búfer.

`Serial.readBytesUntil()` devuelve el número de caracteres leídos en el búfer. Un 0 significa que el parámetro de longitud <= 0, se agotó el tiempo de espera antes que cualquier otra entrada o se encontró un carácter de terminación antes que cualquier otra entrada.

`Serial.readBytesUntil()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.readBytesUntil(character, buffer, length)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.
`character`: el personaje a buscar. Tipos de datos permitidos: `char`.
`buffer`: el búfer para almacenar los bytes. Tipos de datos permitidos: array de `char` o `byte`.
`length`: el número de bytes a leer. Tipos de datos permitidos: `int`.

Devuelve:

Tipo de datos: `size_t`.

Notas y advertencias:

El carácter terminador se descarta del búfer en serie, a menos que el número de caracteres leídos y copiados en el búfer sea igual a `length`.

2.13.1.17. Serial.readString()

Descripción:

`Serial.readString()` lee caracteres del búfer en serie en una cadena. La función finaliza si se agota el tiempo de espera (ver `setTimeout()`).

`Serial.readString()` hereda de la clase de utilidad Stream..

Sintaxis:

```
Serial.readString()
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de Serial.

Devuelve:

Una cadena leída del búfer serial

Código de ejemplo:

Demostrar `Serial.readString()`

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

void loop() {
  Serial.println("Introducir datos:");
  while (Serial.available() == 0) {}     //esperar datos disponibles
  String teststr = Serial.readString();  //leer hasta el tiempo de
  // espera
  teststr.trim();  // elimine cualquier espacio blanco \r \n
  // al final de la cadena
  if (teststr == "rojo") {
    Serial.println("Un color primario");
  } else {
    Serial.println("Otra cosa");
  }
}
```

Notas y advertencias:

La función no finaliza antes si los datos contienen caracteres de final de línea. El `String` devuelto puede contener caracteres de retorno de carro y/o avance de línea si se recibieron.

2.13.1.18. Serial.readStringUntil()

Descripción:

`readStringUntil()` lee caracteres del búfer serial en una cadena. La función finaliza si se agota el tiempo de espera (ver `setTimeout()`).

`Serial.readStringUntil()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.readStringUntil(terminador)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal Serie.
`terminador`: el carácter a buscar. Tipos de datos permitidos: `char`.

Devuelve:

La cadena completa se lee desde el búfer en serie, hasta el carácter terminador.

Notas y advertencias:

El carácter terminador se descarta del búfer serial.

2.13.1.19. Serial.setTimeout()

Descripción:

`Serial.setTimeout()` establece el máximo de milisegundos para esperar datos en serie. El valor predeterminado es 1000 milisegundos.

`Serial.setTimeout()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Serial.setTimeout(time)
```

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.
`time`: duración del tiempo de espera en milisegundos. Tipos de datos permitidos: `long`.

Devuelve:

Nada

Notas y advertencias:

Funciones Serial que utilizan el valor de tiempo de espera establecido a través de `Serial.setTimeout()`:

- `Serial.find()`
- `Serial.findUntil()`
- `Serial.parseInt()`
- `Serial.parseFloat()`
- `Serial.readBytes()`
- `Serial.readBytesUntil()`
- `Serial.readString()`
- `Serial.readStringUntil()`

2.13.1.20. Serial.write()

Descripción:

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

Sintaxis:

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

Parámetros:

`Serial`: objeto de puerto serie. Consulte la lista de puertos serie disponibles para cada placa en la página principal de serie.

`val`: un valor para enviar como un solo byte.
`str`: una cadena para enviar como una serie de bytes.
`buf`: una matriz para enviar como una serie de bytes.
`len`: el número de bytes que se enviarán desde la matriz.

Devuelve:

`write()` devolverá el número de bytes escritos, aunque leer ese número es opcional. Tipo de datos: `size_t`.

Código de ejemplo:

```
void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.write(45); // enviar un byte con el valor 45
  int bytesSent = Serial.write("hello"); //envía la cadena "hola" y
  //devuelve la longitud de la cadena.
}
```

Notas y advertencias:

A partir de Arduino IDE 1.0, la transmisión en serie es asíncrona. Si hay suficiente espacio vacío en el búfer de transmisión, `Serial.write()` regresará antes de que se transmita cualquier carácter a través de la serie. Si el búfer de transmisión está lleno, `Serial.write()` se bloqueará hasta que haya suficiente espacio en el búfer. Para evitar el bloqueo de llamadas a `Serial.write()`, primero puede verificar la cantidad de espacio libre en el búfer de transmisión usando `availableForWrite()`.

2.13.1.21. serialEvent()

Descripción:

Llamado al final de `loop()` cuando los datos están disponibles. Use `Serial.read()` para capturar estos datos.

Sintaxis:

```
void serialEvent() {
  //declaraciones
}
```

Para placas con puertos serie adicionales (consulte la lista de puertos serie disponibles para cada placa en la página principal de serie):

```
void serialEvent1() {
  //declaraciones
}
void serialEvent2() {
  //declaraciones
}
void serialEvent3() {
  //declaraciones
}
```

Parámetros:

`statements`: cualquier declaración válida

Devuelve:

Nada.

Notas y advertencias:

`serialEvent()` No funciona en Leonardo, Micro, o Yún.
`serialEvent()` y `serialEvent1()` no funciona en placas Arduino SAMD
`serialEvent()`, serialEvent1(), serialEvent2(), y serialEvent3() no funcionan en Arduino Due.

2.13.2. SPI

Descripción:

Esta biblioteca le permite comunicarse con dispositivos SPI, con Arduino como dispositivo controlador. Esta biblioteca se incluye con todas las plataformas Arduino (avr, megaavr, mbed, samd, sam, arc32), por lo que **no necesita instalar** la biblioteca por separado.

Para utilizar esta biblioteca:

```
#include <SPI.h>
```

Para leer más sobre Arduino y SPI, puede visitar la guía Arduino & Serial Peripheral Interface (SPI).

2.13.2.1. SPISettings

Descripción:

El objeto `SPISettings` se usa para configurar el puerto SPI para su dispositivo SPI. Los 3 parámetros se combinan en un solo objeto `SPISettings`, que se asigna a `SPI.beginTransaction()`.

Cuando todas sus configuraciones son constantes, `SPISettings` debe usarse directamente en `SPI.beginTransaction()`. Consulte la sección de sintaxis a continuación. Para las constantes, esta sintaxis da como resultado un código más pequeño y más rápido.

Si alguna de sus configuraciones son variables, puede crear un objeto `SPISettings` para contener las 3 configuraciones. Luego puede dar el nombre del objeto a `SPI.beginTransaction()`. La creación de un objeto `SPISettings` con nombre puede ser más eficiente cuando sus configuraciones no son constantes, especialmente si la velocidad máxima es una variable calculada o configurada, en lugar de un número que ingresa directamente en su boceto.

Sintaxis:

```
SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0))
```
Nota: mejor si las 3 configuraciones son constantes
```
SPISettings mySetting(speedMaximum, dataOrder, dataMode)
```
Nota: mejor cuando cualquier configuración es una variable''

Parámetros:

`speedMaximum`: La máxima velocidad de comunicación. Para un chip SPI clasificado hasta 20 MHz, use 20000000.
`dataOrder`: MSBFIRST or LSBFIRST
`dataMode`: SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3

Devuelve:

Nada.

2.13.2.2. SPI.begin()

Descripción:

Inicializa el bus SPI configurando SCK, MOSI y SS en las salidas, bajando SCK y MOSI y SS alto.

Sintaxis:

```
SPI.begin()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

2.13.2.3. SPI.end()

Descripción:

Deshabilita el bus SPI (dejando los modos pin sin cambios).

Sintaxis:

```
SPI.end()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

2.13.2.4. SPI.beginTransaction()

Descripción:

Inicializa el bus SPI utilizando los `SPISettings` definidos.
Sintaxis:

```
SPI.beginTransaction(mySettings)
```

Parámetros:

`mySettings`: la configuración elegida según `SPISettings`.

Devuelve:

Nada.

2.13.2.5. SPI.endTransaction()

Deja de usar el bus SPI. Normalmente, esto se llama después de desactivar la selección de chip, para permitir que otras bibliotecas usen el bus SPI.

Sintaxis:

```
SPI.endTransaction()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

2.13.2.6. SPI.setBitOrder()

Descripción:

Esta función no debe utilizarse en nuevos proyectos. Utilice `SPISettings.` con `SPI.beginTransaction().` para configurar los parámetros SPI.

Establece el orden de los bits desplazados fuera y dentro del bus SPI, ya sea LSBFIRST (el bit menos significativo primero) o MSBFIRST (el bit más significativo primero).

Sintaxis:

```
SPI.setBitOrder(order)
```

Parámetros:

`order`: ya sea `LSBFIRST` o `MSBFIRST`.

Devuelve:

Nada.

2.13.2.7. SPI.setClockDivider()

Descripción:

Esta función no debe utilizarse en nuevos proyectos. Utilice `SPISettings`. con `SPI.beginTransaction()`. para configurar los parámetros SPI.

Establece el divisor de reloj SPI relativo al reloj del sistema. En las placas basadas en AVR, los divisores disponibles son 2, 4, 8, 16, 32, 64 o 128. La configuración predeterminada es SPI_CLOCK_DIV4, que establece el reloj SPI en un cuarto de la frecuencia del reloj del sistema (4 Mhz para las placas a 16 MHz).

Para Arduino Due: en Due, el reloj del sistema se puede dividir por valores de 1 a 255. El valor predeterminado es 21, lo que establece el reloj en 4 MHz como otras placas Arduino.

Sintaxis:

```
SPI.setClockDivider(divider)
```

Parámetros:

`divider` (solo placas AVR):

- `SPI_CLOCK_DIV2`
- `SPI_CLOCK_DIV4`
- `SPI_CLOCK_DIV8`
- `SPI_CLOCK_DIV16`
- `SPI_CLOCK_DIV32`
- `SPI_CLOCK_DIV64`
- `SPI_CLOCK_DIV128`

`chipSelectPin:` pin CS del dispositivo periférico (solo Arduino Due) divisor: un número del 1 al 255 (solo Arduino Due)

Devuelve:

Nada.

2.13.2.8. SPI.setDataMode()

Descripción:

Esta función no debe utilizarse en nuevos proyectos. Utilice `SPISettings`. con `SPI.beginTransaction()` para configurar los parámetros SPI.

Establece el modo de datos SPI: es decir, la polaridad y la fase del reloj. Consulte el artículo de Wikipedia sobre SPI para obtener más detalles.

Sintaxis:

```
SPI.setDataMode(mode)
```

Parámetros:

Modo:

- SPI_MODE0
- SPI_MODE1
- SPI_MODE2
- SPI_MODE3

chipSelectPin: pin CS del dispositivo periférico (solo Arduino Due)

Devuelve:

Nada.

2.13.2.9. SPI.transfer()

Descripción:

La transferencia SPI se basa en un envío y recepción simultáneos: los datos recibidos se devuelven en receiveVal (o receiveVal16). En el caso de transferencias de búfer, los datos recibidos se almacenan en el búfer en el lugar (los datos antiguos se reemplazan con los datos recibidos).

Sintaxis:

```
receivedVal = SPI.transfer(val)
receivedVal16 = SPI.transfer16(val16)
SPI.transfer(buffer, size)
```

Parámetros:

- val: el byte a enviar por el bus
- val16: la variable de dos bytes para enviar por el bus
- buffer: el array de datos a transferir

Devuelve:

Los datos recibidos.

2.13.2.10. SPI.usingInterrupt()

Descripción:

Si su programa realizará transacciones SPI dentro de una interrupción, llame a esta función para registrar el número o nombre de la interrupción con la biblioteca SPI. Esto permite que `SPI.beginTransaction()` evite conflictos de uso. Tenga en cuenta que la interrupción especificada en la llamada a `usingInterrupt()` se deshabilitará en una llamada a `beginTransaction()` y se volverá a habilitar en `endTransaction()`.

Sintaxis:

```
SPI.usingInterrupt(interruptNumber)
```

Parámetros:

`interruptNumber`: el número de interrupción asociado.

Devuelve:

Nada.

2.13.3. Stream

Descripción:

Stream es la clase base para flujos basados en caracteres y binarios. No se llama directamente, pero se invoca cada vez que usa una función que se basa en él.

Stream define las funciones de lectura en Arduino. Al usar cualquier funcionalidad central que use un método `read()` o similar, puede asumir con seguridad que llama a la clase Stream. Para funciones como `print()`, Stream hereda de la clase Print.

Algunas de las bibliotecas que confían en Stream incluyen:

- Serial
- Wire
- Ethernet
- SD

2.13.3.1. Stream.available()

Descripción:

`available()` obtiene el número de bytes disponibles en el flujo. Esto es solo para los bytes que ya han llegado.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.available()
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.

Devuelve:

El número de bytes disponibles para leer. Tipo de datos: `int`.

2.13.3.2. Stream.read()

Descripción:
`read()` lee caracteres de un flujo entrante al búfer.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase de transmisión para obtener más información.

Sintaxis:

```
stream.read()
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.

Devuelve:

El primer byte de datos entrantes disponibles (o -1 si no hay datos disponibles).

2.13.3.3. Stream.flush()

Descripción:

`flush()` borra el búfer una vez que se han enviado todos los caracteres salientes.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase de transmisión para obtener más información.

Sintaxis:

```
stream.flush()
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.

Devuelve:

Nada.

2.13.3.4. Stream.find()

Descripción:

`find()` lee los datos de la secuencia hasta que se encuentra el destino. La función devuelve verdadero si se encuentra el objetivo, falso si se agotó el tiempo (ver `Stream.setTimeout()`).

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase de transmisión para obtener más información.

Sintaxis:

```
stream.find(target)
stream.find(target, length)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`target`: la cadena a buscar. Tipos de datos permitidos: `char`.
`length`: longitud del objetivo. Tipos de datos permitidos: `size_t`.

Devuelve:

Tipo de datos: `bool`.

2.13.3.5. Stream.findUntil()

Descripción:

`findUntil()` lee los datos de la transmisión hasta que se encuentra la cadena de destino de la longitud dada o la cadena de terminación, o se agota el tiempo de espera (ver `Stream.setTimeout()`).

La función devuelve `true` si se encuentra la cadena de destino, `false` si se agotó el tiempo de espera.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de Stream class para obtener más información.

Sintaxis:

```
stream.findUntil(target, terminal)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`target`: la cadena a buscar. Tipos de datos permitidos: `char`.
`terminal`: la cadena terminal en la búsqueda. Tipos de datos permitidos: `char`.

Devuelve:

Tipo de dato: `bool`.

2.13.3.6. Stream.peek()

Descripción:

Lee un byte del archivo sin avanzar al siguiente. Es decir, las llamadas sucesivas a `peek()` devolverán el mismo valor, al igual que la siguiente llamada a `read()`.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.peek()
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.

Devuelve:

El siguiente byte (o carácter), o -1 si no hay ninguno disponible.

2.13.3.7. Stream.readBytes()

Descripción:

`readBytes()` leer caracteres de una secuencia en un búfer. La función finaliza si se ha leído la longitud determinada o se agota el tiempo de espera (ver `setTimeout()`).

`ReadBytes()` devuelve el número de bytes colocados en el búfer. Un 0 significa que no se encontraron datos válidos.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.readBytes(buffer, length)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`buffer`: el búfer para almacenar los bytes. Tipos de datos permitidos: array de `char` o `byte`.
`length`: el número de bytes a leer. Tipos de datos permitidos: `int`.

Devuelve:

El número de bytes colocados en el búfer. Tipo de datos: `size_t`.

2.13.3.8. Stream.readBytesUntil()

Descripción:

`readBytesUntil()` lee caracteres de una secuencia en un búfer. La función finaliza si se detecta el carácter terminador, se ha leído la longitud determinada o se agota el tiempo de espera (ver `setTimeout()`). La función devuelve los caracteres hasta el último carácter antes del terminador proporcionado. El terminador en sí no se devuelve en el búfer.

`readBytesUntil()` devuelve el número de bytes colocados en el búfer. Un 0 significa que no se encontraron datos válidos.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.readBytesUntil(character, buffer, length)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`character`: el carácter a buscar. Tipos de datos permitidos: `char`.
`buffer`: el búfer para almacenar los bytes. Tipos de datos permitidos: array de `char` o `byte` .
`length` : el número de bytes a leer. Tipos de datos permitidos: `int`.

Devuelve:

El número de bytes colocados en el búfer.

Notas y advertencias:

El carácter terminador se descarta del flujo.

2.13.3.9. Stream.readString()

Descripción:

`readString()` lee caracteres de una secuencia en una cadena. La función finaliza si se agota el tiempo de espera (ver `setTimeout()`).

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.readString()
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.

Devuelve:

Una cadena leída de una secuencia.

2.13.3.10. Sream.readStringUntil()

Descripción:

`readStringUntil()` lee caracteres de una secuencia en una cadena. La función finaliza si se detecta el carácter terminador o se agota el tiempo de espera (ver `setTimeout()`).

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.
Sintaxis:

```
stream.readStringUntil(terminador)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`terminador`: el carácter a buscar. Tipos de datos permitidos: `char`.

Devuelve:

Toda la cadena leída de una secuencia, hasta el carácter terminador

Notas y advertencias:

El carácter terminador se descarta del flujo.

2.13.3.11. Stream.parseInt()

Descripción:

`parseInt()` devuelve el primer número entero válido (largo) de la posición actual.

En particular:

- El análisis se detiene cuando no se han leído caracteres para un valor de tiempo de espera configurable, o se lee un dígito que no es.
- Si no se leyeron dígitos válidos cuando se agotó el tiempo de espera (ver `Stream.setTimeout()`), se devuelve 0.

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de `Stream class` para obtener más información.

Sintaxis:

```
stream.parseInt()
stream.parseInt(lookahead)
stream.parseInt(lookahead, ignore)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`lookahead`: el modo utilizado para mirar hacia adelante en la secuencia en busca de un número entero. Tipos de datos permitidos: `LookaheadMode`. Tipos de datos permitidos: `lookahead`:

- `SKIP_ALL`: todos los caracteres que no sean dígitos o un signo menos se ignoran al escanear la secuencia en busca de un número entero. Este es el modo por defecto.
- `SKIP_NONE`: No se salta nada y la secuencia no se toca a menos que el primer carácter de espera sea válido.
- `SKIP_WHITESPACE`: Solo se saltan los tabuladores, los espacios, los saltos de línea y los retornos de carro.

`ignore`: utilizado para omitir el carácter indicado en la búsqueda. Se utiliza, por ejemplo, para omitir el divisor de miles. Tipos de datos permitidos: `char`.

Devuelve:

Tipo de dato: `long`.

2.13.3.12. Stream.parseFloat()

Descripción:

`parseFloat()` devuelve el primer número de punto flotante válido desde la posición actual. `parseFloat()` termina con el primer carácter que no es un número de coma flotante. La función finaliza si se agota el tiempo de espera (ver `Stream.setTimeout()`).

Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.parseFloat()
stream.parseFloat(lookahead)
stream.parseFloat(lookahead, ignore)
```

Parámetros:

`stream` : una instancia de una clase que hereda de Stream.
`lookahead`: el modo utilizado para mirar hacia adelante en la secuencia de un número de coma flotante. Tipos de datos permitidos: `LookaheadMode`. Valores `lookahead` permitidos:

- `SKIP_ALL`: todos los caracteres que no sean el signo menos, el punto decimal o los dígitos se ignoran al escanear la transmisión en busca de un número de coma flotante. Este es el modo por defecto.
- `SKIP_NONE`: No se salta nada y la secuencia no se toca a menos que el primer carácter de espera sea válido.
- `SKIP_WHITESPACE`: Solo se saltan los tabuladores, los espacios, los saltos de línea y los retornos de carro.

`ignore`: utilizado para omitir el carácter indicado en la búsqueda. Se utiliza, por ejemplo, para omitir el divisor de miles. Tipos de datos permitidos: `char`.

Devuelve:

Tipo de dato: `float`.

2.13.3.13. Stream.setTimeout()

Descripción:

`setTimeout()` establece el máximo de milisegundos para esperar la transmisión de datos; el valor predeterminado es 1000 milisegundos. Esta función es parte de la clase Stream y puede ser llamada por cualquier clase que herede de ella (Wire, Serial, etc). Consulte la página principal de la clase Stream para obtener más información.

Sintaxis:

```
stream.setTimeout(time)
```

Parámetros:

`stream`: una instancia de una clase que hereda de Stream.
`time`: duración del tiempo de espera en milisegundos. Tipos de datos permitidos: `long`.

Devuelve:

Nada.

Notas y advertencias:

Funciones de transmisión que usan el valor de tiempo de espera establecido a través de `setTimeout()`:

- `find()`
- `findUntil()`
- `parseInt()`
- `parseFloat()`
- `readBytes()`
- `readBytesUntil()`
- `readString()`
- `readStringUntil()`

2.13.4. Wire

Descripción:

Esta biblioteca le permite comunicarse con dispositivos I2C/TWI. En las placas Arduino con el diseño R3 (pinout 1.0), SDA (línea de datos) y SCL (línea de reloj) están en los encabezados de los pines cerca del pin AREF. El Arduino Due tiene dos interfaces I2C/TWI SDA1 y SCL1 que están cerca del pin AREF y la adicional está en los pines 20 y 21.

Como referencia, la siguiente tabla muestra dónde se encuentran los pines TWI en varias placas Arduino.

Board	I2C/TWI pins
UNO, Ethernet	A4 (SDA), A5 (SCL)
Mega2560	20 (SDA), 21 (SCL)
Leonardo	20 (SDA), 21 (SCL), SDA1, SCL1

A partir de Arduino 1.0, la biblioteca hereda de las funciones Stream, lo que la hace coherente con otras bibliotecas de lectura/escritura. Debido a esto, `send()` y `receive()` han sido reemplazados por `read()` y `write()`.

Las versiones recientes de la biblioteca Wire pueden usar tiempos de espera para evitar un bloqueo ante ciertos problemas en el bus, pero esto no está habilitado de manera predeterminada (todavía) en las versiones actuales. Se recomienda habilitar siempre estos tiempos de espera al usar la biblioteca Wire. Consulte la función `Wire.setWireTimeout` para obtener más detalles.

Nota: Hay versiones de 7 y 8 bits de direcciones I2C. 7 bits identifican el dispositivo, y el octavo bit determina si se está escribiendo o leyendo. La biblioteca Wire utiliza direcciones de 7 bits en todo momento. Si tiene una hoja de datos o un código de muestra que usa una dirección de 8 bits, querrá eliminar el bit bajo (es decir, cambiar el valor un bit a la derecha), lo que genera una dirección entre 0 y 127. Sin embargo, las direcciones de 0 a 7 no se usan porque están reservados, por lo que la primera dirección que se puede usar es 8. Tenga en cuenta que se necesita una resistencia pull-up cuando se conectan los pines SDA/SCL. Consulte los ejemplos para obtener más información. La placa MEGA 2560 tiene resistencias pull-up en los pines 20 y 21 integrados.

La implementación de la biblioteca Wire utiliza un búfer de 32 bytes, por lo que cualquier comunicación debe estar dentro de este límite. Los bytes en exceso en una sola transmisión simplemente se eliminarán.

Para usar esta biblioteca:

```
#include <Wire.h>
```

2.13.4.1. Wire.begin()

Descripción:

Esta función inicializa la biblioteca Wire y se une al bus I2C como controlador o periférico. Esta función normalmente debería llamarse una sola vez.

Sintaxis:

```
Wire.begin()
Wire.begin(address)
```

Parámetros:

`address`: la dirección esclava de 7 bits (opcional); si no se especifica, únase al bus como un dispositivo controlador.

Devuelve:

Nada.

2.13.4.2. Wire.end()

Descripción:

Deshabilite la biblioteca Wire, invirtiendo el efecto de `Wire.begin()`. Para volver a utilizar la biblioteca Wire después de esto, vuelva a llamar a `Wire.begin()`.

Nota: esta función no estaba disponible en la versión original de la biblioteca Wire y es posible que aún no esté disponible en todas las plataformas. El código que debe ser portátil entre plataformas y versiones puede usar la macro `WIRE_HAS_END`, que solo se define cuando `Wire.end()` está disponible.

Sintaxis:

```
Wire.end()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

2.13.4.3. Wire.requestFrom()

Descripción:

El dispositivo controlador utiliza esta función para solicitar bytes de un dispositivo periférico. Luego, los bytes se pueden recuperar con las funciones `available()` y `read()`. A partir de Arduino 1.0.1, `requestFrom()` acepta un argumento booleano que cambia su comportamiento para compatibilidad con ciertos dispositivos I2C. Si es verdadero, `requestFrom()` envía un mensaje de detención después de la solicitud, liberando el bus I2C. Si es falso, `requestFrom()` envía un mensaje de reinicio después de la solicitud. El bus no se liberará, lo que evita que otro dispositivo maestro solicite entre mensajes. Esto permite que un dispositivo maestro envíe múltiples solicitudes mientras tiene el control. El valor por defecto es verdadero.

Sintaxis:

```
Wire.requestFrom(address, quantity)
Wire.requestFrom(address, quantity, stop)
```

Parámetros:

- `address`: la dirección esclava de 7 bits del dispositivo para solicitar bytes.
- `quantity`: el número de bytes a solicitar.
- `stop`: verdadero o falso. true enviará un mensaje de parada después de la solicitud, liberando el bus. False enviará continuamente un reinicio después de la solicitud, manteniendo la conexión activa.

Devuelve:

`byte`: el número de bytes devueltos desde el dispositivo periférico.

2.13.4.4. Wire.beginTransmission()

Descripción:

Esta función inicia una transmisión al dispositivo periférico I2C con la dirección dada. Posteriormente, ponga en cola los bytes para la transmisión con la función `write()` y transmítalos llamando a `endTransmission()`.

Sintaxis:

```
Wire.beginTransmission(address)
```

Parámetros:

`address`: la dirección de 7 bits del dispositivo a transmitir.

Devuelve:

Nada.

2.13.4.5. Wire.endTransmission()

Descripción

Esta función finaliza una transmisión a un dispositivo periférico iniciada por `beginTransmission()` y transmite los bytes que estaban en cola por `write()`. A partir de Arduino 1.0.1, `endTransmission()` acepta un argumento booleano que cambia su comportamiento para que sea compatible con ciertos dispositivos I2C. Si es verdadero, `endTransmission()` envía un mensaje de detención después de la transmisión, liberando el bus I2C. Si es falso, `endTransmission()` envía un mensaje de reinicio después de la transmisión. El bus no se liberará, lo que impide que otro dispositivo controlador transmita entre mensajes. Esto permite que un dispositivo controlador envíe múltiples transmisiones mientras tiene el control. El valor por defecto es verdadero.

Sintaxis

```
Wire.endTransmission()
Wire.endTransmission(stop)
```

Parámetros

`stop`: verdadero o falso. True enviará un mensaje de parada, liberando el bus después de la transmisión. False enviará un reinicio, manteniendo la conexión activa.

Devuelve:

- `0`: éxito.
- `1`: datos demasiado largos para caber en el búfer de transmisión.
- `2`: NACK recibido al transmitir la dirección.
- `3`: NACK recibido en la transmisión de datos.
- `4`: otro error.
- `5`: tiempo de espera

2.13.4.6. Wire.write()

Descripción:

Esta función escribe datos desde un dispositivo periférico en respuesta a una solicitud de un dispositivo controlador, o pone en cola bytes para la transmisión desde un controlador a un dispositivo periférico (entre llamadas a `beginTransmission()` y `endTransmission()`)

Sintaxis:

```
Wire.write(value)
Wire.write(string)
Wire.write(data, length)
```

Parámetros:

`value`: un valor para enviar como un solo byte.
`string`: una cadena para enviar como una serie de bytes.
`data`: una matriz de datos para enviar como bytes.
`length`: el número de bytes a transmitir.

Devuelve

El número de bytes escritos (la lectura de este número es opcional).

Sintaxis:

```
#include <Wire.h>

byte val = 0;

void setup() {
  Wire.begin(); // Únete al autobús I2C
}
void loop() {
  Wire.beginTransmission(44); // Transmitir al dispositivo número 44
  // (0x2C)
  Wire.write(val); // Envía byte de valor
  Wire.endTransmission(); // Deja de transmitir
  val++; // Valor de incremento
  // si alcanza la posición 64 (max)
  if(val == 64) {
    val = 0; // Empezar de nuevo desde el valor más bajo
  }
  delay(500);
}
```

2.13.4.7. Wire.available()

Descripción:

Esta función devuelve el número de bytes disponibles para recuperar con `read()`. Esta función debe llamarse en un dispositivo controlador después de una llamada a `requestFrom()` o en un periférico dentro del controlador `onReceive()`.`available()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Wire.available()
```

Parámetros:

Ninguno.

Devuelve:

El número de bytes disponibles para lectura.

2.13.4.8. Wire.read()

Descripción:

Esta función lee un byte que se transmitió desde un dispositivo periférico a un dispositivo controlador después de una llamada a `requestFrom()` o se transmitió desde un dispositivo controlador a un dispositivo periférico. `read()` hereda de la clase de utilidad Stream.

Sintaxis:

```
Wire.read()
```

Parámetros:

Ninguno.

Devuelve:

El siguiente byte recibido.

Código de ejemplo:

```
#include <Wire.h>

void setup() {
  Wire.begin(); // Únase al bus I2C (la dirección es
  // opcional para el dispositivo controlador)
  Serial.begin(9600); // Iniciar serial para salida
}
void loop() {
  Wire.requestFrom(2, 6); // Solicite 6 bytes del dispositivo
  // esclavo número dos
  // El esclavo puede enviar menos de lo solicitado
  while(Wire.available()) {
    char c = Wire.read(); // Recibir un byte como carácter
    Serial.print(c); // Imprime el carácter
  }
  delay(500);
}
```

2.13.4.9. Wire.setClock()

Descripción:

Esta función modifica la frecuencia del reloj para la comunicación I2C. Los dispositivos periféricos I2C no tienen una frecuencia de reloj de trabajo mínima, sin embargo, 100 KHz suele ser la línea de base.

Sintaxis:

```
Wire.setClock(clockFrequency)
```

Parámetros:

`clockFrequency`*:* el valor (en Hertz) del reloj de comunicación deseado. Los valores aceptados son 100000 (modo estándar) y 400000 (modo rápido). Algunos procesadores también admiten 10000 (modo de baja velocidad), 1000000 (modo rápido plus) y 3400000 (modo de alta velocidad). Consulte la documentación específica del procesador para asegurarse de que se admita el modo deseado.

Devuelve:

Nada.

2.13.4.10. Wire.onReceive()

Descripción:

Esta función registra una función que se llamará cuando un dispositivo periférico reciba una transmisión desde un dispositivo controlador.

Sintaxis:

```
Wire.onReceive(handler)
```

Parámetros:

`handler`: la función a llamar cuando el dispositivo periférico recibe datos; esto debería tomar un solo parámetro int (la cantidad de bytes leídos del dispositivo controlador) y no devolver nada.

Devuelve:

Nada.

2.13.4.11. Wire.onRequest()

Descripción:

Esta función registra una función que se llamará cuando un dispositivo controlador solicite datos de un dispositivo periférico.

Sintaxis:

```
Wire.onRequest(handler)
```

Parámetros:

`handler`: la función a llamar, no toma parámetros y no devuelve nada.

Devuelve:

Nada.

2.13.4.12. Wire.setWireTimeout()

Descripción:

Establece el tiempo de espera para las transmisiones por cable en modo maestro.

Nota: estos tiempos de espera son casi siempre una indicación de un problema subyacente, como dispositivos que funcionan mal, ruido, blindaje insuficiente u otros problemas eléctricos. Estos tiempos de espera evitarán que su boceto se bloquee, pero no resolverán estos problemas. En tales situaciones, a menudo (también) habrá corrupción de datos que no da como resultado un tiempo de espera u otro error y permanece sin ser detectado. Entonces, cuando ocurre un tiempo de espera, es probable que algunos datos leídos o escritos previamente también estén dañados. Es posible que se necesiten medidas adicionales para detectar dichos problemas de manera más confiable (por ejemplo, sumas de verificación o lectura de valores escritos) y recuperarse de ellos (por ejemplo, reinicio completo del sistema). Este tiempo de espera y tales medidas adicionales deben verse como una última línea de defensa, cuando sea posible, la causa subyacente debe corregirse en su lugar.

Sintaxis:

```
Wire.setWireTimeout(timeout, reset_on_timeout)
Wire.setWireTimeout()
```

Parámetros:

`timeout a timeout`: tiempo de espera en microsegundos, si es cero, la verificación de tiempo de espera está deshabilitada.
`reset_on_timeout`: si es cierto, el hardware de Wire se restablecerá automáticamente cuando se agote el tiempo de espera.

Cuando se llama a esta función sin parámetros, se configura un tiempo de espera predeterminado que debería ser suficiente para evitar bloqueos en una configuración típica de maestro único.

Devuelve:

Nada.

Código de ejemplo:

```
#include <Wire.h>

void setup() {
  Wire.begin(); // unirse al bus i2c (dirección opcional para maestro)
  #if defined(WIRE_HAS_TIMEOUT)
    Wire.setWireTimeout(3000 /* us */, true /* reset_on_timeout */);
  #endif
}

byte x = 0;
```

```
void loop() {

  /* Primero, envía un comando al otro dispositivo */

  Wire.beginTransmission(8); // transmitir al dispositivo #8
  Wire.write(123); // enviar comando
  byte error = Wire.endTransmission(); // ejecutar transacción
  if (error) {
    Serial.println("Ocurrió un error al escribir");
    if (error == 5)
      Serial.println("fue un tiempo de espera");
  }

  delay(100);

  /* Luego, lee el resultado */

  #if defined(WIRE_HAS_TIMEOUT)
  Wire.clearWireTimeoutFlag();
  #endif
  byte len = Wire.requestFrom(8, 1); // solicitar 1 byte del
  //dispositivo #8
  if (len == 0) {
    Serial.println("Ocurrió un error al leer");
    #if defined(WIRE_HAS_TIMEOUT)
    if (Wire.getWireTimeoutFlag())
      Serial.println("It was a timeout");
    #endif
    }
  delay(100);
}
```

Notas y advertencias:

La forma en que se implementa este tiempo de espera puede variar entre diferentes plataformas, pero normalmente se activa una condición de tiempo de espera cuando se espera que se complete (una parte de) la transacción (por ejemplo, esperar a que el bus vuelva a estar disponible, esperar un bit ACK o tal vez esperar para que se complete toda la transacción).

Cuando se produce una condición de tiempo de espera de este tipo, la transacción se cancela y `endTransmission()` o `requestFrom()` devolverán un código de error o cero bytes, respectivamente. Si bien esto no resolverá el problema del bus por sí solo (es decir, no eliminará un cortocircuito), al menos evitará el bloqueo potencialmente indefinido y permitirá que su software detecte y tal vez resuelva esta condición.

Si `reset_on_timeout` se estableció en verdadero y la plataforma lo admite, el hardware de Wire también se restablece, lo que puede ayudar a eliminar cualquier estado incorrecto dentro del módulo de hardware de Wire. Por ejemplo, en la plataforma AVR, esto puede ser necesario para reiniciar las comunicaciones después de un tiempo de espera inducido por ruido.

Cuando se activa un tiempo de espera, se establece un indicador que se puede consultar con `getWireTimeoutFlag()` y se debe borrar manualmente usando `clearWireTimeoutFlag()` (y también se borra cuando se llama a `setWireTimeout()`).

Tenga en cuenta que este tiempo de espera también puede activarse mientras espera que el reloj se estire o espera que un segundo maestro complete su transacción. Así que asegúrese de adaptar el tiempo de espera para adaptarse a esos casos si es necesario. Un tiempo de espera típico sería de 25 ms (que es la extensión máxima del reloj permitida por el protocolo SMBus), pero los valores (mucho) más cortos

generalmente también funcionarán.

Notas de portabilidad:

Esta función no estaba disponible en la versión original de la biblioteca Wire y es posible que aún no esté disponible en todas las plataformas. El código que debe ser portátil entre plataformas y versiones puede usar la macro `WIRE_HAS_TIMEOUT`, que solo se define cuando están disponibles `Wire.setWireTimeout()`, `Wire.getWireTimeoutFlag()` y `Wire.clearWireTimeout()`.

Cuando se introdujo esta función de tiempo de espera en la plataforma AVR, inicialmente se mantuvo deshabilitada de forma predeterminada por compatibilidad, esperando que se habilitara en un momento posterior. Esto significa que el valor predeterminado del tiempo de espera puede variar entre (versiones de) plataformas. La configuración de tiempo de espera predeterminada está disponible en la macro `WIRE_DEFAULT_TIMEOUT` y `WIRE_DEFAULT_RESET_WITH_TIMEOUT` Si necesita que se deshabilite el tiempo de espera, se recomienda que lo deshabilite de forma predeterminada usando `setWireTimeout(0)`, aunque actualmente es el valor predeterminado.

2.13.4.13. Wire.clearWireTimeoutFlag()

Descripción:

Borra el indicador de tiempo de espera.

Es posible que los tiempos de espera no estén habilitados de forma predeterminada. Consulte la documentación de `Wire.setWireTimeout()` para obtener más información sobre cómo configurar los tiempos de espera y cómo funcionan.

Sintaxis:

```
Wire.clearTimeout()
```

Parámetros:

Ninguno.

Devuelve:

`bool`: El valor actual de la bandera.

Notas de portabilidad:

Esta función no estaba disponible en la versión original de la biblioteca Wire y es posible que aún no esté disponible en todas las plataformas. El código que debe ser portátil entre plataformas y versiones puede usar la macro `WIRE_HAS_TIMEOUT`, que solo se define cuando están disponibles `Wire.setWireTimeout()`, `Wire.getWireTimeoutFlag()` y `Wire.clearWireTimeout()`.

2.13.4.14. Wire.getWireTimeoutFlag()

Descripción:

Comprueba si se ha producido un tiempo de espera desde la última vez que se borró el indicador.

Este indicador se establece cada vez que se produce un tiempo de espera y se borra cuando se llama a `Wire.clearWireTimeoutFlag()` o cuando se cambia el tiempo de espera mediante `Wire.setWireTimeout()`.

Sintaxis:

```
Wire.getWireTimeoutFlag()
```

Parámetros:

Ninguno.

Devuelve:

`bool`: El valor actual de la bandera.

Notas de portabilidad:

Esta función no estaba disponible en la versión original de la biblioteca Wire y es posible que aún no esté disponible en todas las plataformas. El código que debe ser portátil entre plataformas y versiones puede usar la macro `WIRE_HAS_TIMEOUT`, que solo se define cuando están disponibles `Wire.setWireTimeout()`, `Wire.getWireTimeoutFlag()` y `Wire.clearWireTimeout()`.

2.14. USB

2.14.1. Keyboard

Descripción:

Las funciones del teclado permiten que las placas basadas en micro 32u4 o SAMD envíen pulsaciones de teclas a una computadora conectada a través del puerto USB nativo de su micro.

Nota: No todos los caracteres ASCII posibles, en particular los que no se pueden imprimir, se pueden enviar con la biblioteca del teclado.

La biblioteca admite el uso de teclas modificadoras. Las teclas modificadoras cambian el comportamiento de otra tecla cuando se presionan simultáneamente. Consulte aquí para obtener información adicional sobre las claves admitidas y su uso.

Notas y advertencias:

Estas bibliotecas principales permiten que las placas basadas en 32u4 y SAMD (Leonardo, Esplora, Zero, Due y MKR Family) aparezcan como un mouse y/o teclado nativo en una computadora conectada.

Una advertencia sobre el uso de las librerías de Mouse y Keyboard: si la librería de Mouse o Keyboard se ejecuta constantemente, será difícil programar su placa. Funciones como `Mouse.move()` y

`Keyboard.print()` moverán el cursor o enviarán pulsaciones de teclas a una computadora conectada y solo deben llamarse cuando esté listo para manejarlas. Se recomienda utilizar un sistema de control para activar esta funcionalidad, como un interruptor físico o solo respondiendo a una entrada específica que pueda controlar. Consulte los ejemplos de mouse y teclado para conocer algunas formas de manejar esto.

Cuando use la biblioteca Mouse o Keyboard, puede ser mejor probar su salida primero usando `Serial.print()`. De esta manera, puede estar seguro de que sabe qué valores se informan.

2.14.1.1. Keyboard.begin()

Descripción:

Cuando se usa con una placa Leonardo o Due, `Keyboard.begin()` comienza a emular un teclado conectado a una computadora. Para finalizar el control, use `Keyboard.end()`.

Sintaxis:

```
Keyboard.begin()
Keyboard.begin(layout)
```

Parámetros:

`layout`: la disposición del teclado a utilizar. Este parámetro es opcional y por defecto es `KeyboardLayout_en_US`.

Diseños de teclado:

Actualmente, la biblioteca admite los siguientes diseños de teclado nacionales:

- `KeyboardLayout_de_DE`: Alemania
- `KeyboardLayout_en_US`: USA
- `KeyboardLayout_es_ES`: España
- `KeyboardLayout_fr_FR`: Francia
- `KeyboardLayout_it_IT`: Italia

Devuelve:

Nada.

Código de ejemplo:

```
#include <Keyboard.h>
void setup() {

  // haga que el pin 2 sea una entrada y encienda la  resistencia de
  // pullup por lo que va alta a menos Conectado a tierra:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  // si se presiona el boton
  if (digitalRead(2) == LOW) {
    //Envía el mensaje
    Keyboard.print("Hello!");}}
```

Notas y advertencias:

Los diseños personalizados se pueden crear copiando y modificando un diseño existente. Consulte las instrucciones en el archivo `KeyboardLayout.h` de la biblioteca del teclado.

2.14.1.2. Keyboard.end()

Descripción:

Detiene la emulación del teclado en una computadora conectada. Para iniciar la emulación de teclado, use `Keyboard.begin()`.

Sintaxis:

```
Keyboard.end()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

Código de ejemplo:

```
#include <Keyboard.h>

void setup() {
  //iniciar la comunicación del teclado
  Keyboard.begin();
  //enviar una pulsación de tecla
  Keyboard.print("Hello!");
  //terminar la comunicación del teclado
  Keyboard.end();
}

void loop() {
  //hacer nada
}
```

2.14.1.3. Keyboard.press()

Descripción:

Cuando se llama, `Keyboard.press()` funciona como si se presionara y mantuviera presionada una tecla en su teclado. Útil cuando se usan teclas modificadoras. Para finalizar la pulsación de tecla, utilice `Keyboard.release()` o `Keyboard.releaseAll()`.

Es necesario llamar a `Keyboard.begin()` antes de usar `press()`.

Sintaxis:

```
Keyboard.press(key)
```

Parámetros:

`key`: la tecla a pulsar. Tipos de datos permitidos: `char`.

Devuelve:

Número de pulsaciones de teclas enviadas. Tipo de datos: `size_t`.

Código de ejemplo:

```
#include <Keyboard.h>

// use esta opción para OSX:
char ctrlKey = KEY_LEFT_GUI;
// utilice esta opción para Windows y Linux:
// char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // haga que el pin 2 sea una entrada y encienda el
  // Resistencia de pullup por lo que va alto a menos
  // Conectado a tierra:
  pinMode(2, INPUT_PULLUP);
  // inicializar el control sobre el teclado:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // no hacer nada hasta que el pin 2 esté bajo
    delay(500);
  }
  delay(1000);
  // nuevo documento:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // espere a que se abra una nueva ventana:
  delay(1000);
}
```

2.14.1.4. Keyboard.print()

Descripción:

Envía una o más pulsaciones de teclas a una computadora conectada. Se debe llamar a `Keyboard.print()` después de iniciar `Keyboard.begin()`.

Sintaxis:

```
Keyboard.print(character)
Keyboard.print(characters)
```

Parámetros:

`character`: un char o int para ser enviado a la computadora como una pulsación de tecla.
`characters`: una cadena que se enviará a la computadora como pulsaciones de teclas.

Devuelve:

Número de pulsaciones de teclas enviadas. Tipo de datos: `size_t`.

Código de ejemplos:

```
#include <Keyboard.h>

void setup() {
  // haga que el pin 2 sea una entrada y encienda la
  // Resistencia de pullup por lo que va alta a menos
  // que esté conectado a tierra:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //si se presiona el boton
  if (digitalRead(2) == LOW) {
    //Envía el mensaje
    Keyboard.print("Hello!");
  }
}
```

Notas y advertencias:

Cuando usas el comando `Keyboard.print()`, ¡el Arduino se hace cargo de tu teclado! Asegúrese de tener el control antes de usar el comando. Un botón pulsador para alternar el estado de control del teclado es efectivo.

2.14.1.5. Keyboard.println()

Descripción:

Envía una o más pulsaciones de teclas a un ordenador conectado, seguidas de una pulsación de la tecla Intro.

Se debe llamar a `Keyboard.println()` después de iniciar `Keyboard.begin()`.

Sintaxis:

```
Keyboard.println()
Keyboard.println(character)
Keyboard.println(characters)
```

Parámetros:

`character`: un char o int para ser enviado a la computadora como una pulsación de tecla, seguido de Enter.
`characters`: una cadena que se enviará a la computadora como pulsaciones de teclas, seguida de Enter.

Devuelve:

Número de pulsaciones de teclas enviadas. Tipo de datos: `size_t`.

Código de ejemplo:

```
#include <Keyboard.h>

void setup() {
  // haga que el pin 2 sea una entrada y encienda el
  // Resistencia de pullup por lo que va alta a menos
  // Conectado a tierra:
  pinMode(2, INPUT_PULLUP);
  Keyboard.begin();
}

void loop() {
  //si se presiona el boton
  if (digitalRead(2) == LOW) {
    //Envía el mensaje
    Keyboard.println("Hello!");
  }
}
```

Notas y advertencia:

Cuando usas el comando `Keyboard.println()`, ¡el Arduino se hace cargo de tu teclado! Asegúrese de tener el control antes de usar el comando. Un botón pulsador para alternar el estado de control del teclado es efectivo.

2.14.1.6. Keyboard.release()

Descripción:

Permite ir de la tecla especificada. Consulte `Keyboard.press()` para obtener más información.

Sintaxis:

```
Keyboard.release(key)
```

Parámetros:

`key`: la clave para liberar. Tipos de datos permitidos: `char`.

Devuelve:

El número de llaves liberadas. Tipo de datos: `size_t`.

Código de ejemplo:

```
#include <Keyboard.h>

// use esta opción para OSX:
// char ctrlKey = KEY_LEFT_GUI;
// utilice esta opción para Windows y Linux:
char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // haga que el pin 2 sea una entrada y encienda el
  // Resistencia de pullup por lo que va alta a menos
  // Conectado a tierra:
```

```
  pinMode(2, INPUT_PULLUP);
  // inicializar el control sobre el teclado:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
    // inicializar el control sobre el teclado:
    delay(500);
  }
  delay(1000);
  // nuevo documento:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.release(ctrlKey);
  Keyboard.release('n');
  // espere a que se abra una nueva ventana:
  delay(1000);
}
```

2.14.1.7. Keyboard.releaseAll()

Descripción:

Deja ir todas las teclas actualmente presionadas. Consulte Keyboard.press() para obtener información adicional.

Sintaxis:

```
Keyboard.releaseAll()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

Código de ejemplo:

```
#include <Keyboard.h>
// use esta opción para OSX:
// char ctrlKey = KEY_LEFT_GUI;
// utilice esta opción para Windows y Linux:
char ctrlKey = KEY_LEFT_CTRL;

void setup() {
  // haga que el pin 2 sea una entrada y encienda el
  // Resistencia de pullup por lo que va alta a menos
  // Conectado a tierra:
  pinMode(2, INPUT_PULLUP);
  // inicializar el control sobre el teclado:
  Keyboard.begin();
}

void loop() {
  while (digitalRead(2) == HIGH) {
```

```
    // no hacer nada hasta que el pin 2 esté bajo
    delay(500);
  }
  delay(1000);
  // nuevo documento:
  Keyboard.press(ctrlKey);
  Keyboard.press('n');
  delay(100);
  Keyboard.releaseAll();
  // espere a que se abra una nueva ventana:
  delay(1000);
}
```

2.14.1.8. Keyboard.write()

Descripción:

Envía una pulsación de tecla a una computadora conectada. Esto es similar a presionar y soltar una tecla en su teclado. Puede enviar algunos caracteres ASCII o los modificadores de teclado adicionales y teclas especiales.

Solo se admiten los caracteres ASCII que están en el teclado. Por ejemplo, ASCII 8 (retroceso) funcionaría, pero ASCII 25 (Sustitución) no. Al enviar letras mayúsculas, `Keyboard.write()` envía un comando de cambio más el carácter deseado, como si estuviera escribiendo en un teclado. Si envía un tipo numérico, lo envía como un carácter ASCII (por ejemplo, Keyboard.write(97) enviará 'a').

Para obtener una lista completa de caracteres ASCII, consulte ASCIITable.com.

Sintaxis:

```
Keyboard.write(character)
```

Parámetros:

`character`: un char o int para ser enviado a la computadora. Se puede enviar en cualquier notación que sea aceptable para un char. Por ejemplo, todos los siguientes son aceptables y envían el mismo valor, 65 o ASCII A:

```
Keyboard.write(65); // envía el valor ASCII 65, o A
Keyboard.write('A'); // lo mismo que un caracterer citado
Keyboard.write(0x41); // lo mismo en hexadecimal
Keyboard.write(0b01000001); // lo mismo en binario (elección extraña
// pero funciona)
```

Devuelve:

Número de bytes enviados. Tipo de datos: `size_t`.

Código de ejemplo:

```
#include <Keyboard.h>

void setup() {
  // haga que el pin 2 sea una entrada y encienda el
  // Resistencia de pullup por lo que va alta a menos
  // Conectado a tierra:
  pinMode(2, INPUT_PULLUP);
```

```
  Keyboard.begin();
}

void loop() {
  //si se presiona el boton
  if (digitalRead(2) == LOW) {
  //Enviar un ASCII 'A',
  Keyboard.write(65);
  }
}
```

Notas y advertencias:

Cuando usas el comando `Keyboard.write()`, ¡el Arduino se hace cargo de tu teclado! Asegúrese de tener el control antes de usar el comando. Un botón pulsador para alternar el estado de control del teclado es efectivo.

2.14.2. Mouse

Descripción:

Las funciones del mouse permiten que las placas basadas en micro 32u4 o SAMD controlen el movimiento del cursor en una computadora conectada a través del puerto USB nativo de su micro. Al actualizar la posición del cursor, siempre es relativa a la ubicación anterior del cursor.

Notas y advertencias:

Estas bibliotecas principales permiten que las placas basadas en 32u4 y SAMD (Leonardo, Esplora, Zero, Due y MKR Family) aparezcan como un mouse y/o teclado nativo en una computadora conectada.

Una advertencia sobre el uso de las bibliotecas de Mouse y Keyboard: si la biblioteca de Mouse o Keyboard se ejecuta constantemente, será difícil programar su placa. Funciones como `Mouse.move()` y `Keyboard.print()` moverán el cursor o enviarán pulsaciones de teclas a una computadora conectada y solo deben llamarse cuando esté listo para manejarlas. Se recomienda utilizar un sistema de control para activar esta funcionalidad, como un interruptor físico o solo respondiendo a una entrada específica que pueda controlar. Consulte los ejemplos de mouse y teclado para conocer algunas formas de manejar esto.

Cuando use la biblioteca Mouse o Keyboard, puede ser mejor probar su salida primero usando `Serial.print()`. De esta manera, puede estar seguro de que sabe qué valores se informan.

2.14.2.1. Mouse.begin()

Descripción:

Comienza a emular el mouse conectado a una computadora. `begin()` debe llamarse antes de controlar la computadora. Para finalizar el control, use `Mouse.end()`.

Sintaxis:

```
Mouse.begin()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

Código de ejemplo:

```
#include <Mouse.h>

void setup() {
  pinMode(2, INPUT);
}

void loop() {
  //iniciar la biblioteca del mouse cuando se presiona el botón
  if (digitalRead(2) == HIGH) {
    Mouse.begin();
  }
}
```

2.14.2.2. Mouse.click()

Descripción:

Envía un clic momentáneo a la computadora en la ubicación del cursor. Esto es lo mismo que presionar y soltar inmediatamente el botón del mouse.

`Mouse.click()` por defecto es el botón izquierdo del ratón.

Sintaxis:

```
Mouse.click()
Mouse.click(button)
```

Parámetros:

`button`: qué botón del mouse presionar. Tipos de datos permitidos: `char`.

- `MOUSE_LEFT (default)`
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Devuelve:

Nada.

Código de ejemplo:

```
#include <Mouse.h>

void setup() {
  pinMode(2, INPUT);
  //iniciar la biblioteca de Mouse
  Mouse.begin();
```

```
}
void loop() {
  //si se presiona el botón, envíe un clic izquierdo del mouse
  if (digitalRead(2) == HIGH) {
    Mouse.click();
  }
}
```

Notas y advertencias:

Cuando usas el comando `Mouse.click()` ¡el Arduino se hace cargo de tu mouse! Asegúrese de tener el control antes de usar el comando. Un botón para alternar el estado de control del mouse es efectivo.

2.14.2.3. Mouse.end()

Descripción:

Deja de emular el mouse conectado a una computadora. Para iniciar el control, use `Mouse.begin()`.
Sintaxis:

```
Mouse.end()
```

Parámetros:

Ninguno.

Devuelve:

Nada.

Código de ejemplo:

```
#include <Mouse.h>

void setup() {
  pinMode(2, INPUT);
  //iniciar la biblioteca de Mouse
  Mouse.begin();
}

void loop() {
  //si se presiona el botón, envíe un clic izquierdo del mouse
  //luego finalice la emulación del mouse
  if (digitalRead(2) == HIGH) {
    Mouse.click();
    Mouse.end();
  }
}
```

2.14.2.4. Mouse.move()

Descripción:

Mueve el cursor en una computadora conectada. El movimiento en pantalla siempre es relativo a la ubicación actual del cursor. Antes de usar `Mouse.move()` debe llamar a `Mouse.begin()`.

Sintaxis:

```
Mouse.move(xVal, yVal, wheel)
```

Parámetros:

`xVal`: cantidad para moverse a lo largo del eje x. Tipos de datos permitidos: `signed char`.
`yVal`: cantidad para moverse a lo largo del eje y. Tipos de datos permitidos: `signed char`.
`wheel`: cantidad para mover la rueda de desplazamiento. Tipos de datos permitidos: `signed char`.

Devuelve:

Nada.

Código de ejemplo:

```
#include <Mouse.h>
const int xAxis = A1; // sensor analógico para eje X
const int yAxis = A2; // sensor analógico para eje Y

int range = 12; // rango de salida de movimiento X o Y
int responseDelay = 2; // retardo de respuesta del ratón, en ms
int threshold = range / 4; // umbral de reposo
int center = range / 2; // valor de la posición de reposo
int minima[] = {1023, 1023}; // Mínimos reales de lectura analógica
// para {x, y}
int maxima[] = {0, 0}; // analogRead máximos reales para {x, y}
int axis[] = {xAxis, yAxis}; // números pin para {x, y}
int mouseReading[2]; // lecturas finales del ratón para {x, y}

void setup() {
  Mouse.begin();
}

void loop() {
  // leer y escalar los dos ejes:
  int xReading = readAxis(0);
  int yReading = readAxis(1);
  // mueve el ratón:
  Mouse.move(xReading, yReading, 0);
  delay(responseDelay);
}

/*
  lee un eje (0 o 1 para x o y) y escala el
  rango de entrada analógica a un rango de 0 a <rango>
*/

int readAxis(int axisNumber) {

  int distance = 0; // distancia desde el centro del rango de salida
  // leer la entrada analógica::
  int reading = analogRead(axis[axisNumber]);

  // de la lectura actual excede el máximo o mínimo para este eje,
  // restablecer el máximo o mínimo:
  if (reading < minima[axisNumber]) {
    minima[axisNumber] = reading;
  }
  if (reading > maxima[axisNumber]) {
```

```
    maxima[axisNumber] = reading;
  }
  // asigne la lectura del rango de entrada analógica al rango
  //de salida:
  reading = map(reading, minima[axisNumber], maxima[axisNumber], 0,
  range);

  // si la lectura de salida está fuera del
  // Umbral de posición de reposo, utilícelo:
  if (abs(reading - center) > threshold) {
    distance = (reading – center);
  }

  // el eje Y debe invertirse para mapear el movimiento correctamente:
  if (axisNumber == 1) {
    distance = -distance;
  }
  // devuelve la distancia para este eje:
  return distance;
}
```

Notas y advertencias:

Cuando usas el comando `Mouse.move()`, ¡el Arduino se hace cargo de tu mouse! Asegúrese de tener el control antes de usar el comando. Un botón para alternar el estado de control del mouse es efectivo.

2.14.2.5. Mouse.press()

Descripción:

Envía una pulsación de botón a un ordenador conectado. Una pulsación es el equivalente a hacer clic y mantener presionado continuamente el botón del mouse. Una pulsación se cancela con `Mouse.release()`.

Antes de usar `Mouse.press()`, debe iniciar la comunicación con `Mouse.begin()`.

`Mouse.press()` por defecto es presionar el botón izquierdo.

Sintaxis:

```
Mouse.press()
Mouse.press(button)
```

Parámetros:

`button`: qué botón del ratón presionar. Tipos de datos permitidos: `char`.

- `MOUSE_LEFT (default)`
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Devuelve:

Nada.

Código de ejemplo:

```
#include <Mouse.h>

void setup() {
  //El interruptor que iniciará la pulsación del ratón
  pinMode(2, INPUT);
  //El interruptor que terminará la pulsación del ratón.
  pinMode(3, INPUT);
  //iniciar la biblioteca de Mouse
  Mouse.begin();
}
void loop() {
  //si el interruptor conectado al pin 2 está cerrado, mantenga
  //presionado el botón izquierdo del mouse
  if (digitalRead(2) == HIGH) {
    Mouse.press();
  }
  //si el interruptor conectado al pin 3 está cerrado, suelte el botón
  //izquierdo del mouse
  if (digitalRead(3) == HIGH) {
    Mouse.release();
  }
}
```

Notas y advertencias:

Cuando usas el comando `Mouse.press()`, ¡el Arduino se hace cargo de tu mouse! Asegúrese de tener el control antes de usar el comando. Un botón para alternar el estado de control del mouse es efectivo.

2.14.2.6. Mouse.release()

Descripción:

Envía un mensaje de que se libera un botón previamente presionado (invocado a través de `Mouse.press()`). `Mouse.release()` por defecto es el botón izquierdo.

Sintaxis:

```
Mouse.release()
Mouse.release(button)
```

Parámetros:

`button`: qué botón del ratón presionar. Tipos de datos permitidos: `char`.

- `MOUSE_LEFT` (default)
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Devuelve:

Nada.

Código de ejemplo:

```
#include <Mouse.h>

void setup() {
  //El interruptor que iniciará la pulsación del ratón
  pinMode(2, INPUT);
  //El interruptor que terminará la pulsación del ratón.
  pinMode(3, INPUT);
  //iniciar la biblioteca de Mouse
  Mouse.begin();
}
void loop() {
  //si el interruptor conectado al pin 2 está cerrado, mantenga
  //presionado el botón izquierdo del mouse
  if (digitalRead(2) == HIGH) {
    Mouse.press();
  }
  //si el interruptor conectado al pin 3 está cerrado, suelte el botón
  //izquierdo del mouse
  if (digitalRead(3) == HIGH) {
    Mouse.release();
  }
}
```

Notas y advertencias:

Cuando usas el comando `Mouse.release()`, ¡el Arduino se hace cargo de tu mouse! Asegúrese de tener el control antes de usar el comando. Un botón para alternar el estado de control del mouse es efectivo.

2.14.2.7. Mouse.isPressed()

Descripción:

Comprueba el estado actual de todos los botones del mouse e informa si alguno está presionado o no.

Sintaxis:

```
Mouse.isPressed();
Mouse.isPressed(button);
```

Parámetros:

Cuando no se pasa ningún valor, comprueba el estado del botón izquierdo del ratón.

`button`:qué botón del ratón comprobar. Tipos de datos permitidos: `char`.

- `MOUSE_LEFT` (por defecto)
- `MOUSE_RIGHT`
- `MOUSE_MIDDLE`

Devuelve:

Informa si se presiona o no un botón. Tipo de datos: `bool`.

Código de ejemplo:

```
#include <Mouse.h>

void setup() {
  //El interruptor que iniciará la pulsación del ratón
  pinMode(2, INPUT);
  //El interruptor que terminará la pulsación del ratón.
  pinMode(3, INPUT);
  //Inicie la comunicación serial con la computadora
  Serial.begin(9600);
  //iniciar la biblioteca de Mouse
  Mouse.begin();
}

void loop() {
  //una variable para comprobar el estado del botón
  int mouseState = 0;
  // si el interruptor conectado al pin 2 está cerrado, mantenga
  // presionado el botón izquierdo del mouse y guarde el estado en
  // una variable
  if (digitalRead(2) == HIGH) {
    Mouse.press();
    mouseState = Mouse.isPressed(); }
  //si el interruptor conectado al pin 3 está cerrado, suelte el botón
  //izquierdo del mouse y guarde el estado en una variable
  if (digitalRead(3) == HIGH) {
    Mouse.release();
    mouseState = Mouse.isPressed();}
  //imprimir el estado actual del botón del mouse
  Serial.println(mouseState);
  delay(10);}
```

3. Variables

3.1. Constantes

Las constantes son expresiones predefinidas en el lenguaje Arduino. Se utilizan para facilitar la lectura de los programas. Clasificamos las constantes en grupos:

3.1.1. HIGH|LOW

Al leer o escribir en un pin digital, solo hay dos valores posibles que un pin puede tomar/establecer: `HIGH` y `LOW`.

HIGH

El significado de `HIGH` (en referencia a un pin) es algo diferente dependiendo de si un pin está configurado como `INPUT` o `OUTPUT`. Cuando un pin se configura como `INPUT` con `pinMode()` y se lee con `digitalRead()`, el Arduino (ATmega) informará `HIGH` si:

- un voltaje mayor a 3.0V está presente en el pin (tableros de 5V)
- un voltaje superior a 2,0 V está presente en el pin (placas de 3,3 V)

Un pin también puede configurarse como una `INPUT` con `pinMode()` y, posteriormente, convertirse en `HIGH` con `digitalWrite()`. Esto habilitará las resistencias pull-up internas de 20K, que elevarán el pin de entrada a una lectura `HIGH` a menos que un circuito externo lo baje. Esto se puede hacer alternativa mente pasando `INPUT_PULLUP` como argumento a la función `pinMode()`, como se explica con más detalle en la sección "Definición de los modos de pines digitales: `INPUT`, `INPUT_PULLUP` y `OUTPUT`" más adelante.

Cuando un pin se configura en `OUTPUT` con `pinMode()` y se establece en `HIGH` con `digitalWrite()`, el pin está en:

- 5 voltios (tableros de 5V)
- 3,3 voltios (placas de 3,3 V)

En este estado puede generar corriente, ej. enciende un LED que está conectado a tierra a través de una resistencia en serie.

LOW

El significado de `LOW` también tiene un significado diferente dependiendo de si un pin está configurado en `INPUT` o `OUTPUT`. Cuando un pin se configura como `INPUT` con `pinMode()` y se lee con `digitalRead()`, el Arduino (ATmega) informará `LOW` si:

- un voltaje inferior a 1,5 V está presente en el pin (tableros de 5 V)
- hay un voltaje inferior a 1,0 V (aprox.) en el pin (placas de 3,3 V)

Cuando un pin se configura en `OUTPUT` con `pinMode()` y se establece en `LOW` con `digitalWrite()`, el pin está a 0 voltios (tanto en las placas de 5 V como de 3,3 V). En este estado puede absorber corriente, ej. enciende un LED que está conectado a través de una resistencia en serie a +5 voltios (o +3,3 voltios).

3.1.2. INPUT|OUTPUT|INPUT_PULLUP

Los pines digitales se pueden usar como `INPUT` , `INPUT_PULLUP` o `OUTPUT` . Cambiar un pin con `pinMode()` cambia el comportamiento eléctrico del pin.

Pines configurados como ENTRADA

Se dice que los pines de Arduino (ATmega) configurados como `INPUT` con `pinMode()` están en un estado de *alta impedancia*. Los pines configurados como `INPUT` hacen demandas extremadamente pequeñas en el circuito que están muestreando, equivalentes a una resistencia en serie de 100 megaohmios frente al pin. Esto los hace útiles para leer un sensor.

Si tiene su pin configurado como una `INPUT` y está leyendo un interruptor, cuando el interruptor está en estado abierto, el pin de entrada estará "flotando", lo que dará como resultado resultados impredecibles. Para asegurar una lectura correcta cuando el interruptor está abierto, se debe usar una resistencia pull-up o pull-down. El propósito de esta resistencia es llevar el pin a un estado conocido cuando el interruptor está abierto. Por lo general, se elige una resistencia de 10 K ohmios, ya que es un valor lo suficientemente bajo para evitar de manera confiable una entrada flotante y, al mismo tiempo, un valor lo suficientemente alto para no consumir demasiada corriente cuando el interruptor está cerrado. Consulte el tutorial Digital Read Serial para obtener más información.

Si se usa una resistencia pull-down, el pin de entrada será `LOW` cuando el interruptor esté abierto y `HIGH` cuando el interruptor esté cerrado.

Si se usa una resistencia pull-up, el pin de entrada será `HIGH` cuando el interruptor esté abierto y `LOW` cuando el interruptor esté cerrado.

Pins configurados como INPUT_PULLUP

El microcontrolador ATmega en Arduino tiene resistencias pull-up internas (resistencias que se conectan internamente a la alimentación) a las que puede acceder. Si prefiere usar estos en lugar de resistencias pull-up externas, puede usar el argumento `INPUT_PULLUP` en `pinMode()`.

Consulte el tutorial `Input Pullup Serial` para ver un ejemplo de esto en uso.

Los pines configurados como entradas con `INPUT` o `INPUT_PULLUP` pueden dañarse o destruirse si se conectan a voltajes subterráneos (voltajes negativos) o por encima del riel de alimentación positivo (5V o 3V).

Pines configurados como OUTPUT

Se dice que los pines configurados como `OUTPUT` con `pinMode()` están en un estado de *baja impedancia*. Esto significa que pueden proporcionar una cantidad sustancial de corriente a otros circuitos. Los pines ATmega pueden generar (proporcionar corriente) o hundir (absorber corriente) hasta 40 mA (miliamperios) de corriente a otros dispositivos/circuitos. Esto los hace útiles para alimentar LED porque los LED suelen usar menos de 40 mA. Las cargas superiores a 40 mA (por ejemplo, motores) requerirán un transistor u otro circuito de interfaz.

Los pines configurados como salidas pueden dañarse o destruirse si están conectados a tierra o a los rieles de alimentación positivos.

3.1.3. LED_BUILTIN

La mayoría de las placas Arduino tienen un pin conectado a un LED integrado en serie con una resistencia. La constante `LED_BUILTIN` es el número del pin al que está conectado el LED integrado. La mayoría de las placas tienen este LED conectado al pin digital 13.

3.1.4. true|false

Hay dos constantes que se utilizan para representar la verdad y la falsedad en el lenguaje Arduino: `true` y `false`.

true

A menudo se dice que `true` se define como 1, lo cual es correcto, pero true tiene una definición más amplia. Cualquier número entero que no sea cero es verdadero, en un sentido booleano. Entonces, -1, 2 y -200 también se definen como verdaderos en un sentido booleano.

Tenga en cuenta que las constantes `true` y `false` se escriben en minúsculas a diferencia de `HIGH`, `LOW`, `INPUT` y `OUTPUT`.

False

`false` es el más fácil de definir de los dos. `false` se define como 0 (cero).

3.1.5. Constante de punto flotante (float)

Descripción:

Al igual que las constantes enteras, las constantes de punto flotante se utilizan para hacer que el código sea más legible. Las constantes de punto flotante se intercambian en tiempo de compilación por el valor al que se evalúa la expresión.

Código de ejemplo:

```
float n = 0.005; // 0.005 es una constante de punto flotante
```

Notas y advertencias:

constantes de punto flotante también se pueden expresar en una variedad de notación científica. 'E' y 'e' se aceptan como indicadores de exponente válidos.

CONSTANTE DE COMA FLOTANTE	EVALUA A:	TAMBIÉN EVALUA A:
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	0.000000000067

3.1.6. Constantes enteras

Descripción:

Las constantes enteras son números que se utilizan directamente en un boceto, como 123. De forma predeterminada, estos números se tratan como `int`, pero puede cambiar esto con los modificadores U y L (consulte a continuación).

Normalmente, las constantes enteras se tratan como enteros de base 10 (decimales), pero se puede usar una notación especial (formateadores) para ingresar números en otras bases.

BASE	EJEMPLO	FORMATEADOR	COMENTARIO
10 (decimal)	123	ninguno	
2 (binary)	0b1111011	"0b" inicial	Caracteres 0 y 1 válidos
8 (octal)	0173	"0" inicial	Caracteres 0-7 válidos
16 (hexadecimal)	0x7B	"0x" inicial	Caracteres 0-9, A-F, a-f válidos

Decimales (base 10)

Esta es la matemática de sentido común con la que está familiarizado. Se supone que las constantes sin otros prefijos están en formato decimal.

Código de ejemplo:

```
n = 101; // igual que 101 decimal ((1 * 10^2) + (0 * 10^1) + 1)
```

Binario (base 2)

Solo los caracteres 0 y 1 son válidos.

Código de ejemplo:

```
n = 0b101; // igual que 5 decimales ((1 * 2^2) + (0 * 2^1) + 1)
```

Octal (base 8)

Solo los caracteres del 0 al 7 son válidos. Los valores octales se indican con el prefijo "0" (cero).

Código de ejemplo:

```
n = 0101; // igual que 65 decimal ((1 * 8^2) + (0 * 8^1) + 1)
```

Es posible generar un error difícil de encontrar al incluir (involuntariamente) un cero inicial antes de una constante y hacer que el compilador interprete involuntariamente su constante como octal.

Hexadecimal (base 16)

Los caracteres válidos son del 0 al 9 y las letras de la A a la F; A tiene el valor 10, B es 11, hasta F, que es 15. Los valores hexadecimales se indican con el prefijo "0x". Tenga en cuenta que A-F puede ser mayúscula (A-F) o minúscula (a-f).

Código de ejemplo:

```
n = 0x101; // igual que 257 decimal ((1 * 16^2) + (0 * 16^1) + 1)
```

Notas y advertencias:

Formateadores U & L:

De forma predeterminada, una constante entera se trata como un int con las limitaciones correspondientes en los valores. Para especificar una constante entera con otro tipo de datos, siga con:

- una 'u' o 'U' para forzar la constante en un formato de datos sin firmar. Ejemplo: 33u
- una 'l' o 'L' para forzar la constante en un formato de datos largo. Ejemplo: 100000L
- una 'ul' o 'UL' para forzar la constante en una constante larga sin signo. Ejemplo: 32767ul

3.2. Tipos de datos

3.2.1. array

Descripción:

Una matriz o array es una colección de variables a las que se accede con un número de índice. Los arrays en el lenguaje de programación C++ en los que se escriben los sketches de Arduino pueden ser complicados, pero el uso de arrays simples es relativamente sencillo.

Crear (declarar) un array

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

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[5] = {2, 4, -8, 3, 2};
char message[6] = "hello";
```

Puede declarar una matriz sin inicializarla como en myInts. En myPins declaramos una matriz sin elegir explícitamente un tamaño. El compilador cuenta los elementos y crea una matriz del tamaño apropiado.

Finalmente, puede inicializar y dimensionar su matriz, como en mySensVals. Tenga en cuenta que al declarar una matriz de tipo char, se requiere un elemento más que su inicialización para contener el carácter nulo requerido.

Acceso a una matriz

Las matrices están indexadas a cero, es decir, en referencia a la inicialización de la matriz anterior, el primer elemento de la matriz está en el índice 0, por lo tanto:

```
mySensVals[0] == 2, mySensVals[1] == 4,etc.
```

También significa que en una matriz con diez elementos, el índice nueve es el último elemento.

Por lo tanto:

```
int myArray[10]={9, 3, 2, 4, 3, 2, 7, 8, 9, 11};
// myArray[9] contiene 11
```

```
// myArray[10] no es válido y contiene información aleatoria
// (otra dirección de memoria)
```

Por esta razón, debe tener cuidado al acceder a las matrices. Acceder más allá del final de una matriz (usando un número de índice mayor que el tamaño de matriz declarado - 1) es leer de la memoria que está en uso para otros fines. La lectura de estas ubicaciones probablemente no hará mucho, excepto generar datos no válidos. Escribir en ubicaciones de memoria aleatorias es definitivamente una mala idea y, a menudo, puede conducir a resultados desagradables, como bloqueos o mal funcionamiento del programa. Esto también puede ser un error difícil de rastrear.

A diferencia de BASIC o JAVA, el compilador de C++ no verifica si el acceso a la matriz está dentro de los límites legales del tamaño de la matriz que ha declarado.

Para asignar un valor a una matriz:

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

Para recuperar un valor de una matriz:

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

Matrices y bucles FOR

Las matrices a menudo se manipulan dentro de los bucles, donde el contador de bucles se usa como índice para cada elemento de la matriz. Por ejemplo, para imprimir los elementos de una matriz en el puerto serie, podría hacer algo como esto:

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

Código de ejemplo:

Para ver un programa completo que demuestra el uso de matrices, consulte el (ejemplo de coche fantástico) de los (Tutoriales).

3.2.2. bool

Descripción:

Un `bool` contiene uno de dos valores, `true` o `false`. (Cada variable `bool` ocupa un byte de memoria).

Sintaxis:

```
bool var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

Código de ejemplo:

Este código muestra cómo usar el tipo de datos `bool`.

```
int LEDpin = 5; // LED en pin 5
int switchPin = 13; // interruptor momentáneo en 13, otro lado
// conectado a tierra
bool running = false;

void setup() {
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH); // encienda la resistencia pull-up
}
void loop() {
  if (digitalRead(switchPin) == LOW) {
    // el interruptor está presionado - pullup mantiene el pin alto
    // normalmente
    delay(100); // retardo para interruptor de rebote
    running = !running; // alternar la variable en ejecución
    digitalWrite(LEDpin, running); // indicar mediante LED
  }
}
```

3.2.3. boolean

Descripción:

`boolean` es un alias de tipo no estándar para `bool` definido por Arduino. En su lugar, se recomienda utilizar el `bool` de tipo estándar, que es idéntico.

3.2.4. byte

Descripción:

Un byte almacena un número sin signo de 8 bits, de 0 a 255.

Sintaxis:

```
byte var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

3.2.5. char

Descripción:

Un tipo de datos utilizado para almacenar un valor de carácter. Los caracteres literales se escriben entre comillas simples, así: 'A' (para varios caracteres, cadenas, use comillas dobles: "ABC").

Sin embargo, los caracteres se almacenan como números. Puede ver la codificación específica en el gráfico ASCII. Esto significa que es posible hacer operaciones aritméticas con caracteres, en las que se utiliza el valor ASCII del carácter (por ejemplo, 'A' + 1 tiene el valor 66, ya que el valor ASCII de la letra A mayúscula es 65). Consulte la referencia de `Serial.println` para obtener más información sobre cómo los

caracteres se traducen a números.

El tamaño del tipo de datos `char` es de al menos 8 bits. Se recomienda usar `char` solo para almacenar caracteres. Para un tipo de datos de un byte (8 bits) sin signo, utilice el tipo de datos `byte`.

Sintaxis:

```
char var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

Código de ejemplo:

```
char myChar = 'A';
char myChar = 65; // ambos son equivalentes
```

3.2.6. double

Descripción:

Número de coma flotante de doble precisión. En Uno y otras placas basadas en ATMEGA, ocupa 4 bytes. Es decir, la implementación doble es exactamente igual que la flotante, sin ganar en precisión.

En Arduino Due, los dobles tienen una precisión de 8 bytes (64 bits).

Sintaxis:

```
double var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

Notas y advertencias:

Los usuarios que toman prestado código de otras fuentes que incluye variables dobles pueden desear examinar el código para ver si la precisión implícita es diferente de la que realmente se logra en Arduinos basados en ATMEGA.

3.2.7. float

Descripción:

Tipo de datos para números de punto flotante, un número que tiene un punto decimal. Los números de punto flotante a menudo se usan para aproximar valores analógicos y continuos porque tienen una resolución mayor que los números enteros. Los números de punto flotante pueden ser tan grandes como 3,4028235E+38 y tan bajos como -3,4028235E+38. Se almacenan como 32 bits (4 bytes) de información.

Sintaxis:

```
float var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor que le asigne a esa variable.

Código de ejemplo:

```
float myfloat;
float sensorCalbrate = 1.117;

int x;
int y;
float z;

x = 1;
y = x / 2; // y ahora contiene 0, ints no puede contener fracciones
z = (float)x / 2.0; // z ahora contiene .5 (tienes que usar 2.0, no 2)
```

Notas y advertencias:

Si hace operaciones matemáticas con flotantes, debe agregar un punto decimal; de lo contrario, se tratará como un int. Vea la página de constantes de punto flotante para más detalles.

El tipo de datos flotante tiene solo 6-7 dígitos decimales de precisión. Eso significa el número total de dígitos, no el número a la derecha del punto decimal. A diferencia de otras plataformas, donde puede obtener más precisión utilizando un doble (por ejemplo, hasta 15 dígitos), en Arduino, el doble tiene el mismo tamaño que el flotador.

Los números de punto flotante no son exactos y pueden producir resultados extraños cuando se comparan. Por ejemplo, 6.0/3.0 puede no ser igual a 2.0. En su lugar, debe verificar que el valor absoluto de la diferencia entre los números sea menor que un número pequeño. La conversión de punto flotante a números enteros resulta en truncamiento:

```
float x = 2.9; // Una variable de tipo flotante
int y = x; // 2
```

Si, en cambio, desea redondear durante el proceso de conversión, debe agregar 0.5:

```
float x = 2.9;
int y = x + 0.5; // 3
```

o usa la función `round()`:

```
float x = 2.9;
int y = round(x); // 3
```

Las matemáticas de punto flotante también son mucho más lentas que las matemáticas de números enteros en la realización de cálculos, por lo que deben evitarse si, por ejemplo, un ciclo tiene que ejecutarse a la máxima velocidad para una función de tiempo crítica. Los programadores a menudo hacen todo lo posible para convertir los cálculos de coma flotante en matemáticas enteras para aumentar la velocidad.

3.2.8. int

Descripción:

Los números enteros son su tipo de datos principal para el almacenamiento de números.

En Arduino Uno (y otras placas basadas en ATmega), un int almacena un valor de 16 bits (2 bytes). Esto produce un rango de -32,768 a 32,767 (valor mínimo de -2^15 y valor máximo de (2^15) – 1). En las placas basadas en Arduino Due y SAMD (como MKR1000 y Zero), un int almacena un valor de 32 bits (4 bytes). Esto produce un rango de -2,147,483,648 a 2,147,483,647 (valor mínimo de -2^31 y valor máximo de (2^31) – 1).

int almacena números negativos con una técnica llamada (matemáticas de complemento a 2). El bit más alto, a veces denominado bit de "signo", marca el número como un número negativo. El resto de los bits se invierten y se suma 1.

El Arduino se encarga de manejar los números negativos por usted, para que las operaciones aritméticas funcionen de manera transparente de la manera esperada. Sin embargo, puede haber una complicación inesperada al tratar con el operador de desplazamiento de bits a la derecha (`>>`).

Sintaxis:

```
int var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor que le asigne a esa variable.

Código de ejemplo:

Este código crea un número entero llamado 'countUp', que inicialmente se establece como el número 0 (cero). La variable sube en 1 (uno) cada lazo, mostrándose en el monitor serial.

```
int countUp = 0; //crea una variable entera llamado 'countUp'

void setup() {
  Serial.begin(9600); // utilice el puerto serie para imprimir el número
}
void loop() {
  countUp++; // Agrega 1 al int countUp en cada ciclo
  Serial.println(countUp); // imprime el estado actual de countUp
  delay(1000);
}
```

Notas y advertencias:

Cuando se hace que las variables con signo excedan su capacidad máxima o mínima, *se desbordan.* El resultado de un desbordamiento es impredecible, por lo que debe evitarse. Un síntoma típico de un desbordamiento es el "vuelco" de la variable desde su capacidad máxima hasta su mínima o viceversa, pero no siempre es así. Si desea este comportamiento, use `unsigned int`.

3.2.9. long

Descripción:

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

Si hace operaciones matemáticas con números enteros, al menos uno de los valores debe ser de tipo largo, ya sea una constante entera seguida de una L o una variable de tipo largo, lo que obliga a que sea largo. Vea la página de Constantes enteras para más detalles.

Sintaxis:

```
long var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor asignado a la variable.

Código de ejemplo:

```
long speedOfLight_km_s = 300000L; // consulte la página de constantes
//enteras para obtener una explicación de la 'L'
```

3.2.10. short

Descripción:

Un `short` es un tipo de datos de 16 bits.

En todos los Arduinos (basados en ATMega y ARM), un short almacena un valor de 16 bits (2 bytes). Esto produce un rango de -32,768 a 32,767 (valor mínimo de -2^15 y valor máximo de (2^15) – 1).

Sintaxis:

```
short var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor que le asigne a esa variable.

Código de ejemplo:

```
short ledPin = 13
```

3.2.11. size_t

Descripción:

`size_t` es un tipo de datos capaz de representar el tamaño de cualquier objeto en bytes. Ejemplos del uso de `size_t` son el tipo de retorno de `sizeof()` y `Serial.print()`.

Sintaxis:

```
size_t var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

3.2.12. string

Descripción:

Las cadenas de texto se pueden representar de dos maneras. puede usar el tipo de datos String, que es parte del núcleo a partir de la versión 0019, o puede hacer una cadena a partir de una matriz de tipo char y terminarla en cero. Esta página describe el último método. Para obtener más detalles sobre el objeto String, que le brinda más funcionalidad a costa de más memoria, consulte la página del objeto String.

Sintaxis:

Todas las siguientes son declaraciones válidas para cadenas.

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

Posibilidades para declarar cadenas

- Declare una matriz de caracteres sin inicializarla como en Str1
- Declare una matriz de caracteres (con un carácter adicional) y el compilador agregará el carácter nulo requerido, como en Str2
- Agregue explícitamente el carácter nulo, Str3
- Inicialice con una constante de cadena entre comillas; el compilador dimensionará la matriz para que se ajuste a la constante de cadena y un carácter nulo de terminación, Str4
- Inicialice la matriz con un tamaño explícito y una constante de cadena, Str5
- Inicialice la matriz, dejando espacio adicional para una cadena más grande, Str6

Terminación nula

Generalmente, las cadenas terminan con un carácter nulo (código ASCII 0). Esto permite que las funciones (como `Serial.print()`) indiquen dónde está el final de una cadena. De lo contrario, continuarían leyendo los bytes de memoria subsiguientes que en realidad no forman parte de la cadena.

Esto significa que su cadena debe tener espacio para un carácter más que el texto que desea que contenga. Es por eso que Str2 y Str5 deben tener ocho caracteres, aunque "arduino" solo tiene siete: la última posición se llena automáticamente con un carácter nulo. Str4 se dimensionará automáticamente a ocho caracteres, uno para el valor nulo adicional. En Str3, hemos incluido explícitamente el carácter nulo (escrito `'\0'`) nosotros mismos.

Tenga en cuenta que es posible tener una cadena sin un carácter nulo final (por ejemplo, si hubiera especificado la longitud de Str2 como siete en lugar de ocho). Esto romperá la mayoría de las funciones que usan cadenas, por lo que no debe hacerlo intencionalmente. Sin embargo, si nota que algo se comporta de manera extraña (operando en caracteres que no están en la cadena), este podría ser el problema.

¿Comillas simples o comillas dobles?

Las cadenas siempre se definen dentro de comillas dobles ("Abc") y los caracteres siempre se definen dentro de comillas simples (‘A’).

Envolver cuerdas largas

Puede envolver cadenas largas como esta:

```
char myString[] = "This is the first line"
" this is the second line"
" etcetera";
```

Arrays de cadenas

A menudo es conveniente, cuando se trabaja con grandes cantidades de texto, como un proyecto con una pantalla LCD, configurar una matriz de cadenas. Debido a que las cadenas en sí mismas son matrices, este es en realidad un ejemplo de una matriz bidimensional.

En el siguiente código, el asterisco después del tipo de datos char "char*" indica que se trata de una matriz de "punteros". Todos los nombres de arreglos son en realidad punteros, por lo que esto es necesario para crear un arreglo de arreglos. Los punteros son una de las partes más esotéricas de C ++ para que los principiantes entiendan, pero no es necesario comprender los punteros en detalle para usarlos de manera efectiva aquí.

Código de ejemplo:

```
char *myStrings[] = {"This is string 1", "This is string 2", "This is
                     string 3", "This is string 4", "This is string 5",
                     "This is string 6"
                     };
void setup() {
  Serial.begin(9600);
}
void loop() {
  for (int i = 0; i < 6; i++) {
  Serial.println(myStrings[i]);
  delay(500);
 }
}
```

3.2.13. String()

Descripción:

Construye una instancia de la clase String. Hay varias versiones que construyen cadenas a partir de diferentes tipos de datos (es decir, les dan formato como secuencias de caracteres), que incluyen:

- una cadena constante de caracteres, entre comillas dobles (es decir, una matriz de caracteres)
- un único carácter constante, entre comillas simples
- otra instancia del objeto String
- un entero constante o entero largo
- un entero constante o un entero largo, utilizando una base especificada
- una variable entera o entera larga
- una variable entera o entera larga, utilizando una base especificada
- un flotante o doble, usando un lugar decimal especificado

La construcción de una cadena a partir de un número da como resultado una cadena que contiene la representación ASCII de ese número. El valor predeterminado es base diez, por lo que

```
String thisString = String(13);
```

te da la cadena "13". Sin embargo, puedes usar otras bases. Por ejemplo,

```
String thisString = String(13, HEX);
```

le da la cadena "d", que es la representación hexadecimal del valor decimal 13. O si lo prefiere binario,

```
String thisString = String(13, BIN);
```

le da la cadena "1101", que es la representación binaria de 13.

Sintaxis:

```
String(val)
String(val, base)
String(val, decimalPlaces)
```

Parámetros:

`val`: una variable para formatear como una cadena. Tipos de datos permitidos: `string`, `char`, `byte`, `int`, `long`, `unsigned int`, `unsigned long`, `float`, `double`.
`base`: (opcional) la base en la que dar formato a un valor integral.
`DecimalPlaces`: **solo si val es float o double**. Los lugares decimales deseados.

Devuelve:

Una instancia de la clase String.

Código de ejemplo:

Todas las siguientes son declaraciones válidas para Strings.

```
String stringOne = "Hello String"; // usando una cadena constante
String stringOne = String('a'); // convertir un carácter constante
// en una cadena
String stringTwo = String("This is a string"); // convertir una cadena
// constante en un objeto String
String stringOne = String(stringTwo + " with more"); // concatenar dos
// cadenas
String stringOne = String(13); // utilizando un entero constante
String stringOne = String(analogRead(0), DEC); // usando un int y una
// base
String stringOne = String(45, HEX); // usando un int y una base
// (hexadecimal)
String stringOne = String(255, BIN); // usando un int y una base
// (binario)
String stringOne = String(millis(), DEC); // Usando un long y una base
String stringOne = String(5.698, 3); // usando un float y los
// lugares decimales
```

3.2.13.1. Funciones

3.2.13.1.1. charAt()

Descripción:

Accede a un carácter particular del String.

Sintaxis:

```
miCadena.charAt(n)
```

Parámetros:

`myString`: una variable de tipo `String`.
n: una variable. Tipos de datos permitidos: `unsigned int`.

Devuelve:

El carácter en el índice n de String.

3.2.13.1.2. compareTo()

Descripción:

Compara dos cadenas, probando si una viene antes o después de la otra, o si son iguales. Las cadenas se comparan carácter por carácter, utilizando los valores ASCII de los caracteres. Eso significa, por ejemplo, que 'a' viene antes de 'b' pero después de 'A'. Los números van antes que las letras.

Sintaxis:

```
miCadena.compareTo(miCadena2)
```

Parámetros:

`myString`: una variable de tipo String.
`myString2`: otra variable de tipo String.

Devoluciones:

`un número negativo`: si myString viene antes de myString2.
`0`: si String es igual a myString2.
`un número positivo`: si myString viene después de myString2.

3.2.13.1.3. concat()

Descripción:

Agrega el parámetro a una cadena.

Sintaxis:

```
myString.concat(parámetro)
```

Parámetros:

`myString`: una variable de tipo `String`.
`parámetro`: Tipos de datos permitidos: `String, string, char, byte, int, unsigned int, long, unsigned long, float, doble, __FlashStringHelper(F() macro).`

Devuelve:

`true`: éxito.
`false`: falla (en cuyo caso la cadena no cambia).

3.2.13.1.4. c_str()

Descripción:

Convierte el contenido de una cadena en una cadena terminada en cero de estilo C. Tenga en cuenta que esto da acceso directo al búfer de cadenas interno y debe usarse con cuidado. En particular, nunca debe modificar la cadena a través del puntero devuelto. Cuando modifica el objeto String, o cuando se destruye, cualquier puntero devuelto previamente por c_str() se vuelve inválido y no debe usarse más.

Sintaxis:

```
myString.c_str()
```

Parámetros:

`myString`: una variable de tipo String.

Devuelve:

Un puntero a la versión de estilo C de la cadena de invocación.

3.2.13.1.5. endsWith()

Descripción:

Comprueba si un String termina o no con los caracteres de otro String.

Sintaxis:

```
myString.endsWith(myString2)
```

Parámetros:

`myString`: una variable de tipo `String`.
`myString2`: otra variable de tipo `String`.

Devoluciones:

`true`: si myString termina con los caracteres de myString2.
`false`: de lo contrario.

3.2.13.1.6. equals()

Descripción:

Compara dos cadenas para la igualdad. La comparación distingue entre mayúsculas y minúsculas, lo que significa que la cadena "hola" no es igual a la cadena "HOLA".

Sintaxis:

```
myString.equals(myString2)
```

Parámetros:

`myString`, `myString2`: variables de tipo `String`.

Devoluciones:

`true`: si string es igual a string2.
`falso`: si string NO es iigual a string2.

3.2.13.1.7. equalsIgnoreCase()

Descripción:

Compara dos cadenas para la igualdad. La comparación no distingue entre mayúsculas y minúsculas, lo que significa que String("hola") es igual a String("HOLA").

Sintaxis:

```
miCadena.equalsIgnoreCase(miCadena2)
```

Parámetros:

`myString`: variable de tipo `String`.
`myString2`: variable de tipo `String`.

Devoluciones:

`true`: si myString es igual a myString2 (ignorando mayúsculas y minúsculas).
`falso`: si string NO es iigual a string2.

3.2.13.1.8. getBytes()

Descripción:

Copia los caracteres de String en el búfer suministrado.

Sintaxis:

```
miCadena.getBytes(buf, len)
```

Parámetros:

`myString`: una variable de tipo `String`.
`buf`: el búfer para copiar los caracteres. Tipos de datos permitidos: array de `byte`.
`len`: el tamaño del búfer. Tipos de datos permitidos: `unsigned int`.

Devuelve:

Nada.

3.2.13.1.9. indexOf()

Descripción

Localiza un carácter o una cadena dentro de otra cadena. De forma predeterminada, busca desde el principio de la cadena, pero también puede comenzar desde un índice dado, lo que permite ubicar todas las instancias del carácter o la cadena.

Sintaxis:

```
myString.indexOf(val)
myString.indexOf(val, from)
```

Parámetros:

`myString`: una variable de tipo `String`.
`val`: el valor a buscar. Tipos de datos permitidos: `char`, `String`.
`from`: el índice desde el que comenzar la búsqueda.

Devuelve:

El índice de val dentro de la Cadena, o -1 si no se encuentra.

3.2.13.1.10. lastInddexOf()

Descripción:

Localiza un carácter o una cadena dentro de otro String. De forma predeterminada, busca desde el final del String, pero también puede trabajar hacia atrás desde un índice determinado, lo que permite localizar todas las instancias del carácter o del String.

Sintaxis:

```
myString.lastIndexOf(val)
myString.lastIndexOf(val, from)
```

Parámetros:

`myString`: una variable de tipo `String`.
`val`: el valor a buscar. Tipos de datos permitidos: `char`, `String`.
`from`: el índice desde el que trabajar hacia atrás.

Devuelve:

El índice de val dentro de la Cadena, o -1 si no se encuentra.

3.2.13.1.11. length()

Descripción:

Devuelve la longitud de la Cadena, en caracteres. (Tenga en cuenta que esto no incluye un carácter nulo final).

Sintaxis:

```
miCadena.longitud()
```

Parámetros:

`myString`: una variable de tipo `String`.

Devuelve:

La longitud de la cadena en caracteres. Tipo de datos: `unsigned int`.

3.2.13.1.12. remove()

Descripción:

Modifique en su lugar una Cadena eliminando los caracteres del índice proporcionado hasta el final de la cadena o del índice proporcionado al índice más el conteo.

Sintaxis:

```
myString.remove(índice)
myString.remove(índice, cuenta)
```

Parámetros:

`myString`: una variable de tipo `String`.
`índice`: la posición en la que se inicia el proceso de eliminación (cero indexado). Tipos de datos permitidos: `unsigned int`.
`count`: El número de caracteres a eliminar. Tipos de datos permitidos: `unsigned int`.

Devuelve:

Nada.

Código de ejemplo:

```
Sritng saludo = "hello";
saludo.remove(2, 2); // el saludo ahora contiene "heo"
```

3.2.13.1.13. replace()

Descripción:

La función `String replace()` le permite reemplazar todas las instancias de un carácter dado con otro carácter. También puede usar `replace` para reemplazar subcadenas de una cadena con una subcadena diferente.

Sintaxis:

```
myString.replace(subcadena1, subcadena2)
```

Parámetros:

`myString`: una variable de tipo `String`.
`substring1`: otra variable de tipo `String`.
`substring2`: otra variable de tipo `String`.

Devuelve:

Nada.

3.2.13.1.14. reserve()

Descripción:

La función `String reserve()` le permite asignar un búfer en la memoria para manipular cadenas.

Sintaxis:

```
myString.reserva(tamaño)
```

Parámetros:

`myString`: una variable de tipo `String`.
`tamaño`: el número de bytes en la memoria para guardar para la manipulación de cadenas. Tipos de datos permitidos: `unsigned int`.

Devuelve:

Nada.

Código de ejemplo:

```
String myString;

void setup() {
  // inicialice serial y espere a que se abra el puerto:
  Serial.begin(9600);
  while (! string) {
    ; // espere a que el puerto serie se conecte. Requerido para USB
      // nativo
  }

  myString.reserve(26);
  myString = "i=";
  myString += "1234";
  myString += ", is that ok?";

  // imprime la cadena:
  Serial.println(myString);
}

void loop() {
  // nada que hacer aquí
}
```

3.2.13.1.15. setCharAt()

Descripción:

Establece un carácter de la cadena. No tiene efecto en índices fuera de la longitud existente de String.

Sintaxis:

```
myString.setCharAt(index, c)
```

Parámetros:

`myString`: una variable de tipo `String`.
`index`: el índice para establecer el carácter..
`c`: el carácter para almacenar en la ubicación dada.

Devoluciones:

Nada.

3.2.13.1.16. startsWith()

Descripción:

Comprueba si una cadena comienza o no con los caracteres de otra cadena.

Sintaxis:

```
myString.startsWith(myString2)
```

Parámetros:

`myString, myString2`: una variable de tipo `String`.

Devuelve:

`true`: si myString comienza con los caracteres de myString2.
`false`: si myString NO comienza con los caracteres de myString2.

3.2.13.1.17. substring()

Descripción:

Obtener una subcadena de una Cadena. El índice inicial es inclusivo (el carácter correspondiente se incluye en la subcadena), pero el índice final opcional es exclusivo (el carácter correspondiente no se incluye en la subcadena). Si se omite el índice final, la subcadena continúa hasta el final de la Cadena.

Sintaxis:

```
miCadena.subcadena(from)
myString.substring(from, to)
```

Parámetros:

`myString`: una variable de tipo `String`.
`from`: el índice para comenzar la subcadena.
`to` (opcional): el índice para terminar la subcadena antes.

Devuelve:

La subcadena.

3.2.13.1.18. toCharArray()

Descripción:

Copia los caracteres de String en el búfer suministrado.

Sintaxis:

```
miCadena.toCharArray(buf, len)
```

Parámetros:

`myString`: una variable de tipo `String`.
`buf`: el búfer para copiar los caracteres. Tipos de datos permitidos: matriz de `char`.
`len`: el tamaño del búfer. Tipos de datos permitidos: `unsigned int`.

Devoluciones:

Nada.

3.2.13.1.19. toDouble()

Descripción:

Convierte una cadena válida en un doble. La cadena de entrada debe comenzar con un dígito. Si la cadena contiene caracteres que no son dígitos, la función dejará de realizar la conversión. Por ejemplo, las cadenas "123,45", "123" y "123fish" se convierten en 123,45, 123,00 y 123,00 respectivamente. Tenga en cuenta que "123,456" se aproxima a 123,46. Tenga en cuenta también que los flotantes tienen solo 6-7 dígitos decimales de precisión y que las cadenas más largas pueden truncarse.

Sintaxis:

```
myString.toDouble()
```

Parámetros:

`myString`: una variable de tipo `String`.

Devuelve:

Si no se pudo realizar una conversión válida porque la cadena no comienza con un dígito, se devuelve un cero. Tipo de dato: `doble`.

3.2.13.1.20. toInt()

Descripción:

Convierte una cadena válida en un entero. La cadena de entrada debe comenzar con un número entero. Si la cadena contiene números no enteros, la función dejará de realizar la conversión.

Sintaxis:

```
myString.toInt()
```

Parámetros:

`myString`: una variable de tipo `String`.

Devoluciones:

Si no se pudo realizar una conversión válida porque la cadena no comienza con un número entero, se devuelve un cero. Tipo de dato: `long`.

3.2.13.1.21. toFloat()

Descripción:

Convierte una cadena válida en un flotante. La cadena de entrada debe comenzar con un dígito. Si la cadena contiene caracteres que no son dígitos, la función dejará de realizar la conversión. Por ejemplo, las cadenas "123,45", "123" y "123fish" se convierten en 123,45, 123,00 y 123,00 respectivamente. Tenga en cuenta que "123,456" se aproxima a 123,46. Tenga en cuenta también que los flotantes tienen solo 6-7 dígitos decimales de precisión y que las cadenas más largas pueden truncarse.

Sintaxis:

```
myString.toFloat()
```

Parámetros:

`myString`: una variable de tipo `String`.

Devuelve:

Si no se pudo realizar una conversión válida porque la cadena no comienza con un dígito, se devuelve un cero. Tipo de dato: `float`.

3.2.13.1.22. toLowerCase()

Descripción:

Obtenga una versión en minúsculas de un String. A partir de 1.0, toLowerCase() modifica la cadena en lugar de devolver una nueva.

Sintaxis:

```
myString.toLowerCase()
```

Parámetros:

`myString`: una variable de tipo `String`.

Devoluciones:

Nada.

3.2.13.1.23. toUpperCase()

Descripción:

Obtenga una versión en mayúsculas de un String. A partir de 1.0, `toUpperCase()` modifica la cadena en lugar de devolver una nueva.

Sintaxis:

```
myString.toUpperCase()
```

Parámetros:

`myString`: una variable de tipo `String`.

Devoluciones:

Nada.

3.2.13.1.24. trim()

Descripción:

Obtenga una versión de String sin los espacios en blanco iniciales y finales. A partir de 1.0, `trim()` modifica la cadena en lugar de devolver una nueva.

Sintaxis:

```
myString.trim()
```

Parámetros:

`myString`: una variable de tipo String.

Devuelve:

Nada.

3.2.13.2. Operadores

3.2.13.2.1. [] (element access)

Descripción:

Le permite acceder a los caracteres individuales de una cadena.

Sintaxis:

```
char thisChar = myString1[n]
```

Parámetros:

`thisChar`: Tipos de datos permitidos: `char`.
`myString1`: Tipos de datos permitidos: `String`.
`n`: una variable numérica.

Devoluciones:

El carácter `n` de la cadena. Igual que `charAt()`.

3.2.13.2.2. + (concatenación/enlace)

Descripción:

Combina o concatena dos cadenas en una nueva cadena. El segundo String se agrega al primero y el resultado se coloca en un nuevo String. Funciona igual que `string.concat()`.

Sintaxis:

```
miCadena3 = miCadena1 + miCadena2
```

Parámetros:

`myString1`: una variable de cadena.
`myString2`: una variable de cadena.
`myString3`: una variable de cadena.

Devuelve:

New String que es la combinación de los dos Strings originales.

3.2.13.2.3. += (adjuntar)

Descripción:

Concatena Strings con otros datos.

Sintaxis:

```
myString1 += datos
```

Parámetros:

`myString1`: una variable de cadena.

Devuelve:

Nada.

3.2.13.2.4. == (comparación)

Descripción:

Compara dos cadenas para la igualdad. La comparación distingue entre mayúsculas y minúsculas, lo que significa que la cadena "hola" no es igual a la cadena "HOLA". Funciona igual que `string.equals()`.

Sintaxis:

```
miCadena1 == miCadena2
```

Parámetros:

`myString1`: una variable de cadena.
`myString2`: una variable de cadena.

Devuelve:

`true`: si myString1 es igual a myString2.
`false`: de lo contrario.

3.2.13.2.5. > (mas grande que)

Descripción:

Comprueba si la Cadena de la izquierda es mayor que la Cadena de la derecha. Este operador evalúa Cadenas en orden alfabético, en el primer carácter donde los dos difieren. Entonces, por ejemplo, "b" > "a" y "2" > "1", pero "999" > "1000" porque el 9 viene después del 1.

Precaución: los operadores de comparación de cadenas pueden ser confusos cuando se comparan cadenas numéricas, porque los números se tratan como cadenas y no como números. Si necesita comparar números numéricamente, compárelos como enteros, flotantes o largos, y no como cadenas.

Sintaxis:

```
myString1 > myString2
```

Parámetros:

`myString1`: una variable de cadena.
`myString2`: una variable de cadena.

Devuelve:

`true`: si myString1 es mayor que myString2.
`false`: si myString1 no es mayor que myString2.

3.2.13.2.6. >= (mayor qué o igual que)

Descripción:

Comprueba si la cadena de la izquierda es mayor o igual que la cadena de la derecha. Este operador evalúa cadenas en orden alfabético, en el primer carácter donde los dos difieren. Entonces, por ejemplo, "b" >= "a" y "2" >= "1", pero "999" >= "1000" porque el 9 viene después del 1.

Precaución: los operadores de comparación de cadenas pueden ser confusos cuando se comparan cadenas numéricas, porque los números se tratan como cadenas y no como números. Si necesita comparar números numéricamente, compárelos como enteros, flotantes o largos, y no como cadenas.

Sintaxis:

```
myString1 >= myString2
```

Parámetros:

`myString1`: variable de tipo String.
`myString2`: variable de tipo String.

Devuelve:

`true`: si myString1 es mayor o igual que myString2.
`false`: si myString1 no es mayor o igual que myString2

3.2.13.2.7. < (menor que)

Descripción:

Comprueba si la Cadena de la izquierda es menor que la Cadena de la derecha. Este operador evalúa cadenas en orden alfabético, en el primer carácter donde los dos difieren. Entonces, por ejemplo, "a" < "b" y "1" < "2", pero "999" > "1000" porque el 9 viene después del 1.

Precaución: los operadores de comparación de cadenas pueden ser confusos cuando se comparan cadenas numéricas, porque los números se tratan como cadenas y no como números. Si necesita comparar números numéricamente, compárelos como enteros, flotantes o largos, y no como cadenas.

Sintaxis:

```
myString1 < myString2
```

Parámetros:

`myString1`: variable de tipo String.
`myString2`: variable de tipo String.

Devuelve:

`true`: si myString1 es menor que myString2.
`false`: si myString1 no es menor que myString2.

3.2.13.2.8. <= (Menos que o igual a)

Descripción:

Comprueba si la cadena de la izquierda es menor o igual que la cadena de la derecha. Este operador evalúa cadenas en orden alfabético, en el primer carácter donde los dos difieren. Entonces, por ejemplo, "a" < "b" y "1" < "2", pero "999" > "1000" porque el 9 viene después del 1.

Precaución: los operadores de comparación de cadenas pueden ser confusos cuando se comparan cadenas numéricas, porque los números se tratan como cadenas y no como números. Si necesita comparar números numéricamente, compárelos como enteros, flotantes o largos, y no como cadenas.

Sintaxis:

```
myString1 <= myString2
```

Parámetros:

`myString1`: variable de tipo String.
`myString2`: variable de tipo String.

Devuelve:

`true`: si myString1 es menor o igual que myString2.
`false`: si myString1 no es menor o igual que myString2.

3.2.13.2.9. != (diferente de)

Descripción:

Compara dos cadenas por diferencia. La comparación distingue entre mayúsculas y minúsculas, lo que significa que la cadena "hola" no es igual a la cadena "HOLA". Funcionalmente igual que `string.equals()`

Sintaxis:

```
myString1 != myString2
```

Parámetros:

`myString1`: una variable de cadena.
`myString2`: una variable de cadena.

Devuelve:

`true`: si myString1 es diferente de myString2.
`false`: de lo contrario.

3.2.14. unsigned char

Descripción:

Un tipo de datos unsigned que ocupa 1 byte de memoria. Igual que el tipo de datos `byte`.

El tipo de datos unsigned char codifica números del 0 al 255.

Para mantener la coherencia del estilo de programación de Arduino, se prefiere el tipo de datos de `byte`.

Sintaxis:

```
unsigned char var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

Código de ejemplo:

```
unsigned char myChar = 240;
```

3.2.15. unsigned int

Descripción:

En Uno y otras placas basadas en ATMEGA, los ints sin signo (enteros sin signo) son iguales a los int en el sentido de que almacenan un valor de 2 bytes. Sin embargo, en lugar de almacenar números negativos, solo almacenan valores positivos, lo que genera un rango útil de 0 a 65,535 ((2^16) - 1).

Due almacena un valor de 4 bytes (32 bits), que va de 0 a 4,294,967,295 (2^32 – 1).

La diferencia entre entradas sin signo y entradas (con signo) radica en la forma en que se interpreta el bit más alto, a veces denominado bit de "signo". En el tipo int de Arduino (que está firmado), si el bit alto es un "1", el número se interpreta como un número negativo, y los otros 15 bits se interpretan con (matemáticas de complemento a 2).

Sintaxis:

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

Parámetros:

`var`: nombre de la variable.
`val`: el valor que le asigne a esa variable.

Código de ejemplo:

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

Notas y advertencias:

Cuando se hace que las variables sin signo excedan su capacidad máxima, se "revierten" a 0, y también al revés:

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

Las matemáticas con variables unsigned pueden producir resultados inesperados, incluso si su variable sin signo nunca se da vuelta.

La MCU (microcontroller unit) aplica las siguientes reglas:

El cálculo se realiza en el ámbito de la variable de destino. P.ej. si la variable de destino es signed, hará operaciones matemáticas firmadas, incluso si ambas variables de entrada son unsigned.

Sin embargo, con un cálculo que requiere un resultado intermedio, el código no especifica el alcance del resultado intermedio. En este caso, la MCU hará operaciones matemáticas sin signo para el resultado intermedio, ¡porque ambas entradas no tienen signo!

```
unsigned int x = 5;
unsigned int y = 10;
int result;

result = x - y; // 5 - 10 = -5, como se esperaba
result = (x - y) / 2; // 5 - 10 en matemáticas sin signo es 65530!
// 65530/2 = 32765

// solución: use variables con signo, o haga el cálculo paso a paso.
result = x - y; // 5 - 10 = -5, como se esperaba
result = result / 2; // -5/2 = -2 (solo matemáticas enteras, los
//lugares decimales se eliminan)
```

¿Por qué usar variables unsigned en absoluto?

- Se desea el comportamiento de rollover, p. contadores
- La variable signed es demasiado pequeña, pero desea evitar la pérdida de memoria y velocidad de long/float.

3.2.16. unsigned long

Descripción:

Las variables largas sin signo son variables de tamaño extendido para el almacenamiento de números y almacenan 32 bits (4 bytes). A diferencia de los largos estándar, los unsigned long no almacenan números negativos, por lo que su rango va de 0 a 4,294,967,295 (2^32 – 1).

Sintaxis:

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

Parámetros:

`var`: nombre de la variable.
`val`: el valor que le asigne a esa variable.

Código de ejemplo:

```
unsigned long time;

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

void loop() {
  Serial.print("Time: ");
  time = millis();
  // imprime el tiempo desde que se inició el programa
  Serial.println(time);
  // espera un segundo para no enviar cantidades masivas de datos
  delay(1000);
}
```

3.2.17. void

Descripción:

La palabra clave `void` se usa solo en declaraciones de funciones. Indica que se espera que la función no devuelva información a la función desde la que se llamó.

Código de ejemplo:

```
// las acciones se realizan en las funciones "setup" y "loop"
// pero no se reporta información al programa más grande
void setup() {
// …
}
void loop() {
// …
}
```

3.2.18. word

Descripción:

Una palabra puede almacenar un número sin signo de al menos 16 bits (de 0 a 65535).

Sintaxis:

```
word var = val;
```

Parámetros:

`var`: nombre de la variable.
`val`: el valor a asignar a esa variable.

Código de ejemplo:

```
word w = 10000;
```

3.3. Conversiones

3.3.1. (unsigned int)

Descripción:

Convierte un valor al tipo de datos `unsigned int`.

Sintaxis:

```
(unsigned int)x
```

Parámetros:

`x`: un valor de cualquier tipo

Devuelve:

`unsigned int`.

3.3.2. (unsigned long)

Descripción:

Convierte un valor al tipo de datos `unsigned long`.

Sintaxis:

```
(unsigned long)x
```

Parámetros:

`x`: un valor de cualquier tipo.

Devuelve:

`unsigned long`.

3.3.3. byte()

Descripción:

Convierte un valor al tipo de datos `byte`.

Sintaxis:

```
byte(x)
(byte)x (Conversión de tipo de estilo C)
```

Parámetros:

`x`: un valor. Tipos de datos permitidos: cualquier tipo.

Devuelve:

Tipo de dato: `byte`.

3.3.4. char()

Descripción:

Convierte un valor al tipo de datos `char` .

Sintaxis:

```
char(x)
(char)x (Conversión de tipo de estilo C)
```

Parámetros:

x: un valor. Tipos de datos permitidos: cualquier tipo.

Devuelve:

Tipo de dato: `char`.

3.3.5. float()

Descripción:

Convierte un valor al tipo de datos `float`.

Sintaxis:

```
float(x)
(float)x (Conversión de tipo de estilo C)
```

Parámetros:

x: un valor. Tipos de datos permitidos: cualquier tipo

Devuelve:

Tipo de datos: `float`.

Notas y advertencias:

Consulte la referencia de `float` para obtener detalles sobre la precisión y las limitaciones de los números de punto flotante en Arduino.

3.3.6. int()

Descripción:

Convierte un valor al tipo de datos `int` .

Sintaxis:

```
int(x)
(int)x (Conversión de tipo de estilo C)
```

Parámetros:

x: un valor. Tipos de datos permitidos: cualquier tipo.

Devuelve:

Tipo de dato: `int`.

3.3.7. long()

Descripción:

Convierte un valor al tipo de datos `long`.

Sintaxis:

```
long(x)
long)x (C-style type conversion)
```

Parámetros:

`x`: un valor. Tipos de datos permitidos: cualquier tipo.

Devuelve:

Tipo de dato: `long`.

3.3.8. word

Descripción:

Convierte un valor al tipo de datos `word`.

Sintaxis:

```
word(x)
word (h, l)
(word)x (conversión de tipo de estilo C)
```

Parámetros:

`x`: un valor. Tipos de datos permitidos: cualquier tipo.
`h`: el byte de orden superior (más a la izquierda) de la palabra.
`l`: el byte de orden inferior (más a la derecha) de la palabra.

Devuelve:

Tipo de dato: `word`.

3.4. Alcance variable y calificadores (Variable Scope & Qualifiers)

3.4.1. const

Descripción:

La palabra clave `const` significa constante. Es un *calificador* de variable que modifica el comportamiento de la variable, haciendo que una variable sea de "*solo lectura*". Esto significa que la variable se puede usar como cualquier otra variable de su tipo, pero su valor no se puede cambiar. Obtendrá un error del compilador si intenta asignar un valor a una variable `const` .

Las constantes definidas con la palabra clave `const` obedecen las reglas de variable scoping que rigen otras variables. Esto, y los peligros de usar `#define` , hacen que la palabra clave `const` sea un método superior para definir constantes y se prefiera al uso de `#define`.

Código de ejemplo:

```
const float pi = 3.14;
float x;
// .…
x = pi * 2; // está bien usar constantes en matemáticas
pi = 7; // ilegal: no puede escribir (modificar) una constante
```

Notas y advertencias:

`#define` o `const`

Puede usar `const` o `#define` para crear constantes numéricas o de cadena. Para `arrays`, deberá usar `const`. En general, se prefiere `const` a `#define` para definir constantes.

3.4.2. scope

Descripción:

Las variables en el lenguaje de programación C++, que usa Arduino, tienen una propiedad llamada alcance. Esto contrasta con las primeras versiones de lenguajes como BASIC donde cada variable es una variable *global*.

Una variable global es aquella que puede ser vista por cada función en un programa. Las variables locales solo son visibles para la función en la que se declaran. En el entorno Arduino, cualquier variable declarada fuera de una función (por ejemplo, `setup()`, `loop()`, etc.), es una *variable global.*

Cuando los programas comienzan a hacerse más grandes y complejos, las variables locales son una forma útil de garantizar que solo una función tenga acceso a sus propias variables. Esto evita errores de programación cuando una función modifica inadvertidamente las variables utilizadas por otra función.

A veces también es útil declarar e inicializar una variable dentro de un bucle `for`. Esto crea una variable a la que solo se puede acceder desde dentro de los corchetes del bucle `for`.

Código de ejemplo:

```
int gPWMval; // cualquier función verá esta variable

void setup() {
  // ...
}

void loop() {
  int i; // "i" solo es "visible" dentro de "loop""
  float f; // "f" solo es "visible" dentro de "loop"
  // ...
  for (int j = 0; j < 100; j++) {
    // solo se puede acceder a la variable j dentro de los corchetes
    // del bucle for
  }
}
```

3.4.3. static

Descripción:

La palabra clave `static` se usa para crear variables que son visibles para una sola función. Sin embargo, a diferencia de las variables locales que se crean y destruyen cada vez que se llama a una función, las variables estáticas persisten más allá de la llamada a la función, preservando sus datos entre las llamadas a la función.

Las variables declaradas como estáticas solo se crearán e inicializarán la primera vez que se llame a una función.

Código de ejemplo:

```
/* Caminata aleatoria
  RandomWalk deambula hacia arriba y hacia abajo aleatoriamente entre
  dos puntos finales El movimiento máximo en un ciclo se rige por el
  parámetro "tamaño de paso". Una variable estática se mueve hacia
  arriba y hacia abajo una cantidad aleatoria.
  Esta técnica también se conoce como "ruido rosa" y "caminar
  borracho".
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20

int stepsize;
int thisTime;

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

void loop() {
  //  probar la función RandomWalk
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
  delay(10);
}

int randomWalk(int moveSize) {
  static int place; // variable para almacenar valor en caminata
  // aleatoria declarada estática para que almacene valores entre
  // llamadas de función, pero ninguna otra función puede cambiar su
  // valor

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange) { // comprobar los límites inferior y
  // superior
    place = randomWalkLowRange + (randomWalkLowRange - place);
    // reflejar el número en dirección positiva
  }
  else if (place > randomWalkHighRange) {
    place = randomWalkHighRange - (place - randomWalkHighRange);
    // reflejar el número en dirección negativa
  }

  return place;
}
```

3.4.4. volatile

Descripción:

`volatile` es una palabra clave conocida como *calificador* de variable, generalmente se usa antes del tipo de datos de una variable, para modificar la forma en que el compilador y el programa posterior tratan la variable.

Declarar una variable `volatile` es una directiva para el compilador. El compilador es un software que traduce su código C/C++ al código de la máquina, que son las instrucciones reales para el chip Atmega en el Arduino.

Específicamente, indica al compilador que cargue la variable desde la RAM y no desde un registro de almacenamiento, que es una ubicación de memoria temporal donde se almacenan y manipulan las variables del programa. Bajo ciertas condiciones, el valor de una variable almacenada en registros puede ser inexacto.

Una variable debe declararse `volatile` siempre que su valor pueda ser cambiado por algo más allá del control de la sección de código en la que aparece, como un subproceso que se ejecuta simultáneamente. En Arduino, el único lugar donde es probable que esto ocurra es en secciones de código asociadas con interrupciones, denominadas rutinas de servicio de interrupciones.

int or long volatiles

Si la variable `volatile` es más grande que un byte (por ejemplo, un int de 16 bits o una longitud de 32 bits), el microcontrolador no puede leerlo en un solo paso, porque es un microcontrolador de 8 bits. Esto significa que mientras su sección de código principal (por ejemplo, su bucle) lee los primeros 8 bits de la variable, es posible que la interrupción ya cambie los segundos 8 bits. Esto producirá valores aleatorios para la variable.

Recurso:

Mientras se lee la variable, las interrupciones deben desactivarse, para que no puedan alterar los bits mientras se leen. Hay varias maneras de hacer esto:

1. `noInterrupts (2.11.2)`
2. utilice la macro ATOMIC_BLOCK. Las operaciones atómicas son operaciones MCU individuales: la unidad más pequeña posible.

Código de ejemplo:

El modificador `volatile` garantiza que los cambios en la variable de `state` sean inmediatamente visibles en `loop()`. Sin el modificador `volatile`, la variable de estado puede cargarse en un registro al ingresar a la función y no se actualizará más hasta que finalice la función.

```
// Parpadea el LED durante 1 s si la entrada ha cambiado
// en el segundo anterior.
volatile byte changed = 0;

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(2), toggle, CHANGE);
}
void loop() {
```

```
if (changed == 1) {
// ¡toggle() ha sido llamado desde interrupciones!
// Restablecer cambiado a 0
  changed = 0;
  // LED parpadeante durante 200 ms
  digitalWrite(LED_BUILTIN, HIGH);
  delay(200);
  digitalWrite(LED_BUILTIN, LOW);
 }
}

void toggle() {
  changed = 1;
}
```

Para acceder a una variable con un tamaño mayor que el bus de datos de 8 bits del microcontrolador, use la macro `ATOMIC_BLOCK`. La macro asegura que la variable se lea en una operación atómica, es decir, su contenido no se puede alterar mientras se lee.

```
#include <util/atomic.h> // esta biblioteca incluye la macro
//ATOMIC_BLOCK.

volatile int input_from_interrupt;
// En algún lugar del código, ej. bucle interior()
  ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
  // código con interrupciones bloqueadas (las operaciones atómicas
  // consecutivas no se interrumpirán)
    int result = input_from_interrupt;
}
```

3.5. Utilidades

3.5.1. PROGREM

Descripción:

Almacene datos en la memoria flash (programa) en lugar de SRAM. Hay una descripción de los distintos tipos de memoria disponibles en una placa Arduino.

La palabra clave `PROGMEM` es un modificador de variable, debe usarse solo con los tipos de datos definidos en pgmspace.h. Le dice al compilador que "ponga esta información en la memoria flash", en lugar de en SRAM, donde normalmente iría.

`PROGMEM` es parte de la biblioteca pgmspace.h . Se incluye automáticamente en las versiones modernas del IDE. Sin embargo, si está utilizando una versión IDE anterior a la 1.0 (2011), primero deberá incluir la biblioteca en la parte superior de su boceto, así:

`#include <avr/pgmspace.h>` Si bien `PROGMEM` podría usarse en una sola variable, realmente solo vale la pena si tiene un bloque de datos más grande que necesita almacenarse, lo que generalmente es más fácil en una matriz (o otra estructura de datos de C++ más allá de nuestra presente discusión).

El uso de `PROGMEM` también es un procedimiento de dos pasos. Después de obtener los datos en la memoria Flash, se requieren métodos especiales (funciones), también definidos en la biblioteca `pgmspace.h`, para leer los datos de la memoria del programa nuevamente en SRAM, para que podamos hacer algo útil con ellos.

Sintaxis:

```
const dataType variableName[] PROGMEM = {data0, data1, data3…};
```

Tenga en cuenta que debido a que `PROGMEM` es un modificador de variable, no existe una regla estricta sobre dónde debe ir, por lo que el compilador Arduino acepta todas las definiciones a continuación, que también son sinónimas. Sin embargo, los experimentos han indicado que, en varias versiones de Arduino (que tienen que ver con la versión GCC), `PROGMEM` puede funcionar en una ubicación y no en otra. El ejemplo de "tabla de cadenas" a continuación ha sido probado para funcionar con Arduino 13. Las versiones anteriores del IDE pueden funcionar mejor si se incluye `PROGMEM` después del nombre de la variable.

```
const dataType variableName[] PROGMEM = {}; // usa esta forma
const PROGMEM dataType variableName[] = {}; // o esta
const dataType PROGMEM variableName[] = {}; // esta no
```

Parámetros:

`dataType`: Tipos de datos permitidos: cualquier tipo de variable.
`variableName`: el nombre de su conjunto de datos.

Código de ejemplo:

Los siguientes fragmentos de código ilustran cómo leer y escribir caracteres sin firmar (bytes) e ints (2 bytes) en PROGMEM.

```
// guardar algunos unsigned ints
const PROGMEM uint16_t charSet[] = { 65000, 32796, 16843, 10, 11234};

// guardar algunos caracteres
const char signMessage[] PROGMEM = {"I AM PREDATOR, UNSEEN COMBATANT.
  CREATED BY THE UNITED STATES DEPART"};

unsigned int displayInt;
char myChar;

void setup() {
  Serial.begin(9600);
  while (!Serial); // espere a que se conecte el puerto serie.
  // Necesario para USB nativo
  // ponga su código de configuración aquí, para ejecutar una vez:
  // leer de vuelta un int de 2 bytes
  for (byte k = 0; k < 5; k++) {
    displayInt = pgm_read_word_near(charSet + k);
    Serial.println(displayInt);
  }
  Serial.println();
  // leer un char
  for (byte k = 0; k < strlen_P(signMessage); k++) {
    myChar = pgm_read_byte_near(signMessage + k);
    Serial.print(myChar);
  }
  Serial.println();
}
void loop() {
  // pon tu código principal aquí, para que se ejecute repetidamente:
}
```

Arrays de cadenas

A menudo, cuando se trabaja con grandes cantidades de texto, como un proyecto con una pantalla LCD, es conveniente configurar una matriz de cadenas. Debido a que las cadenas en sí mismas son matrices, este es en realidad un ejemplo de una matriz bidimensional.

Estos tienden a ser estructuras grandes, por lo que a menudo es deseable colocarlos en la memoria del programa. El siguiente código ilustra la idea.

```
/*
PROGMEM string demo
Cómo almacenar una tabla de cadenas en la memoria del programa (flash),
y recuperarlas.

Información resumida de:
http://www.nongnu.org/avr-libc/user-manual/pgmspace.html

Configurar una tabla (matriz) de cadenas en la memoria del programa
es un poco complicado, pero aquí hay una buena plantilla a seguir.

Configurar las cuerdas es un proceso de dos pasos. Primero, defina las
cadenas.
*/

#include <avr/pgmspace.h>
const char string_0[] PROGMEM = "String 0"; // "String 0" etc son
//cadenas para almacenar - cambiar para adaptarse.
const char string_1[] PROGMEM = "String 1";
const char string_2[] PROGMEM = "String 2";
const char string_3[] PROGMEM = "String 3";
const char string_4[] PROGMEM = "String 4";
const char string_5[] PROGMEM = "String 5";

// Luego configure una tabla para referirse a sus cadenas.

const char *const string_table[] PROGMEM = {string_0, string_1,
string_2, string_3, string_4, string_5};

char buffer[30]; // asegúrese de que sea lo suficientemente grande
// para la cuerda más grande que debe contener

void setup() {
  Serial.begin(9600);
  while (!Serial); // espere a que se conecte el puerto serie.
  //Necesario para USB nativo
  Serial.println("OK");
}

void loop() {

/* El uso de la tabla de cadenas en la memoria del programa requiere
el uso de funciones especiales para recuperar los datos. La función
strcpy_P copia una cadena del espacio del programa a una cadena en la
RAM ("búfer").
Asegúrese de que su cadena de recepción en RAM sea lo suficientemente
grande para contener lo que sea está recuperando del espacio del
programa. */

  for (int i = 0; i < 6; i++) {
    strcpy_P(buffer, (char *)pgm_read_word(&(string_table[i])));
    // Moldes necesarios y desreferenciación, solo copiar.
    Serial.println(buffer);
```

```
    delay(500);
  }
}
```

Notas y advertencias:

Tenga en cuenta que las variables deben definirse globalmente o definirse con la palabra clave estática para poder trabajar con `PROGMEM`.

El siguiente código NO funcionará dentro de una función:

```
const char long_str[] PROGMEM = "Hola me gustaria contarte un poco
sobre mí.\n";
```

El siguiente código funcionará, incluso si se define localmente dentro de una función:

```
const static char long_str[] PROGMEM = "hola me gustaria contarte un
un poco sobre mí.\n"
```

La macro `F()`

Cuando una instrucción como:

```
Serial.print("Escribir algo en el monitor serie");
```

se utiliza, la cadena que se va a imprimir normalmente se guarda en la RAM. Si su boceto imprime muchas cosas en el monitor serie, puede llenar fácilmente la memoria RAM. Si tiene espacio libre en la memoria FLASH, puede indicar fácilmente que la cadena se debe guardar en FLASH usando la sintaxis:

```
Serial.print(F("Escriba algo en el Monitor Serial que esté almacenado
en FLASH"));
```

3.5.2. sizeof()

Descripción:

El operador `sizeof` devuelve el número de bytes en un tipo de variable, o el número de bytes ocupados por una matriz.

Sintaxis:

```
sizeof(variable)
```

Parámetros:

`variable`: La cosa para obtener el tamaño de. Tipos de datos permitidos: cualquier tipo de variable o matriz (por ejemplo, `int`, `float`, `byte`).

Devuelve:

El número de bytes en una variable o bytes ocupados en una matriz. Tipo de datos: `size_t`.

Código de ejemplo:

El operador `sizeof` es útil para manejar arreglos (como cadenas) donde es conveniente poder cambiar el tamaño del arreglo sin romper otras partes del programa.

Este programa imprime una cadena de texto de un carácter a la vez. Intente cambiar la frase de texto.

```
char myStr[] = "this is a test";

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

void loop() {
  for (byte i = 0; i < sizeof(myStr) - 1; i++) {
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000); // ralentizar el programa
}
```

Notas y advertencias:

Tenga en cuenta que `sizeof` devuelve el número total de bytes. Entonces, para matrices de tipos de variables más grandes, como ints , el ciclo for se vería así.

```
int myValues[] = {123, 456, 789};

// este ciclo for funciona correctamente con una matriz de cualquier
// tipo o tamaño
for (byte i = 0; i < (sizeof(myValues) / sizeof(myValues[0])); i++)
{
  // hacer algo con myValues[i]
}
```

Tenga en cuenta que una cadena con el formato correcto termina con el símbolo NULL, que tiene el valor ASCII 0.

4. Estructura

4.1. Estructuras de control

4.1.1. if

Descripción:

La declaración `if` verifica una condición y ejecuta la siguiente declaración o conjunto de declaraciones si la condición es 'verdadera'.

Sintaxis:

```
if (condición) {
  //declaracion(es)
}
```

Parámetros:

`condición`: una expresión booleana (es decir, puede ser `true` o `false`).

Código de ejemplo:

Los corchetes pueden omitirse después de una declaración `if`. Si se hace esto, la siguiente línea (definida por el punto y coma) se convierte en la única declaración condicional.

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

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

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

if (x > 120) {
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}
// todas son correctas
```

Notas y advertencias:

Las declaraciones que se evalúan dentro de los paréntesis requieren el uso de uno o más operadores que se muestran a continuación.

Operadores de comparación:

```
x == y (x is equal to 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)
```

Tenga cuidado con el uso accidental del signo igual único (por ejemplo, si `if (x = 10)`). El único signo igual es el operador de asignación y *establece* `x` en 10 (pone el valor 10 en la variable `x`). En su lugar, utilice

el signo igual doble (por ejemplo, si `if (x = 10)`), que es el operador de comparación y *comprueba si* `x` es igual a 10 o no. La última afirmación solo es verdadera si `x` es igual a 10, pero la primera afirmación siempre será verdadera.

Esto se debe a que C++ evalúa la declaración `if (x = 10)` de la siguiente manera: 10 se asigna a `x` (recuerde que el único signo igual es el (operador de asignación)), por lo que `x` ahora contiene 10. Luego, el condicional `if` evalúa 10 , que siempre se evalúa como `true`, ya que cualquier número distinto de cero se evalúa como `true`. En consecuencia, `if (x = 10)` siempre se evaluará como `TRUE`, que no es el resultado deseado cuando se usa una declaración '`if`'. Además, la variable `x` se establecerá en 10, que tampoco es una acción deseada.

4.1.2. else

Descripción:

El `if …else` permite un mayor control sobre el flujo de código que la instrucción `if` básica, al permitir que se agrupen múltiples pruebas. Se ejecutará una cláusula `else` (si es que existe) si la condición en la declaración `if` da como resultado `false`. El `else` puede continuar con otra prueba `if`, de modo que se pueden ejecutar varias pruebas mutuamente excluyentes al mismo tiempo.

Cada prueba pasará a la siguiente hasta que se encuentre una prueba verdadera. Cuando se encuentra una prueba verdadera, se ejecuta su bloque de código asociado y el programa salta a la línea que sigue a toda la construcción `if`/`else`. Si ninguna prueba resulta ser cierta, se ejecuta el bloque `else` predeterminado, si hay uno presente, y establece el comportamiento predeterminado.

Tenga en cuenta que un bloque `else if` puede usarse con o sin un bloque de terminación `else` y viceversa. Se permite un número ilimitado de otras sucursales `else if`.

Sintaxis:

```
if (condición1) {
  // hacer la cosa A
}
else if (condición2) {
  // hacer la cosa B
}
else {
  // hacer la cosa C
}
```

Código de ejemplo:

A continuación se muestra un extracto de un código para el sistema de sensor de temperatura:

```
if (temperature >= 70) {
  // ¡Peligro! Apague el sistema.
}
else if (temperature >= 60) { // 60 <= temperature < 70
  // ¡Advertencia! Se requiere atención del usuario.
}
else { // temperature < 60
  // ¡A salvo! Continuar con las tareas habituales.
}
```

4.1.3. return

Descripción:

Finalice una función y devuelva un valor de una función a la función que llama, si lo desea.

Sintaxis:

```
return;
return value;
```

Parámetros:

`value`: Tipos de datos permitidos: cualquier variable o tipo constante.

Código de ejemplo:

Una función para comparar la entrada de un sensor con un umbral:

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

La palabra clave `return` es útil para probar una sección de código sin tener que "comentar" grandes secciones de código posiblemente con errores.

```
void loop() {
  // brillante idea de código para probar aquí
  return;
  // el resto de un boceto disfuncional aquí
  // este código nunca se ejecutará
}
```

4.1.4. switch...case

Descripción:

Al igual que las declaraciones `if`, `switch...case` controla el flujo de programas al permitir que los programadores especifiquen diferentes códigos que deben ejecutarse en varias condiciones. En particular, una sentencia `switch` compara el valor de una variable con los valores especificados en sentencias case. Cuando se encuentra una declaración de caso cuyo valor coincide con el de la variable, se ejecuta el código en esa declaración de caso.

La palabra clave `break` sale de la instrucción `switch` y normalmente se usa al final de cada caso. Sin una sentencia `break`, la sentencia `switch` continuará ejecutando las siguientes expresiones ("fallingthrough, que cae a través") hasta que se alcance una rotura o el final de la sentencia `switch`.

Sintaxis:

```
switch (var) {
  case label1:
    // declaraciones
    break;
  case label2:
    // declaraciones
    break;
  default:
    // declaraciones
    break;
}
```

Parámetros:

`var` : una variable cuyo valor comparar con varios casos. Tipos de datos permitidos: `int`, `char`.
`label1, label2`: constantes. Tipos de datos permitidos: `int`, `char`.

Devuelve:

Nada.

Código de ejemplo:

```
switch (var) {
  case 1:
    //hacer algo cuando var es igual a 1
    break;
  case 2:
    //hacer algo cuando var es igual a 2
    break;
  default:
    // si nada más coincide, haga lo predeterminado
    // default es opcional
    break;
}
```

4.1.5. while

Descripción:

Un bucle `while` se repetirá de forma continua e infinita hasta que la expresión entre paréntesis, () se vuelva falsa. Algo debe cambiar la variable probada, o el ciclo `while` nunca saldrá. Esto podría estar en su código, como una variable incrementada o una condición externa, como probar un sensor.

Sintaxis:

```
while (condition) {
  // statement(s)
}
```

Parámetros:

`condition` : una expresión booleana que se evalúa como `true` o `false`.

Código de ejemplo:

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

4.1.6. do...while

Descripción:

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

Sintaxis:

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

Parámetros:

`condición` : una expresión booleana que se evalúa como `true` o `false`.

Código de ejemplo:

```
int x = 0;
do {
  delay(50); // esperar a que los sensores se estabilicen
  x = readSensors(); // revisa los sensores
} while (x < 100);
```

4.1.7. for

Descripción:

La declaración `for` se usa para repetir un bloque de declaraciones entre llaves. Normalmente se utiliza un contador de incrementos para incrementar y terminar el bucle. La declaración `for` es útil para cualquier operación repetitiva y, a menudo, se usa en combinación con matrices para operar en colecciones de datos/pins.

Sintaxis:

```
for (inicialización; condición; incremento) {
  // declaraciones;
}
```

Parámetros:

`inicialización`: sucede primero y exactamente una vez.
`condición`: cada vez que se pasa por el ciclo, se prueba la `condición`; si es `true`, se ejecuta el bloque de instrucción y el **incremento**, entonces la **condición** se prueba nuevamente. Cuando la **condición** se vuelve `false`, el ciclo termina.

`incremento`: se ejecuta cada vez que pasa por el ciclo cuando la `condición` es `true`.

Código de ejemplo:

```
// Atenuar un LED usando un pin PWM
int PWMpin = 10; // LED en serie con resistencia de 470 ohmios en el
// pin 10

void setup() {
  // no se necesita configuración
}

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

Notas y advertencias:

El bucle `for` de C++ es mucho más flexible que los bucles `for` que se encuentran en otros lenguajes informáticos, incluido BASIC. Se puede omitir cualquiera de los tres elementos del encabezado o todos ellos, aunque se requieren los puntos y coma. Además, las declaraciones para la inicialización, la condición y el incremento pueden ser cualquier declaración de C++ válida con variables no relacionadas y usar cualquier tipo de datos de C++, incluidos los floats. Estos tipos de declaraciones `for` inusuales pueden proporcionar soluciones a algunos problemas de programación raros.

Por ejemplo, usar una multiplicación en la línea de incremento generará una progresión logarítmica:

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

Genera: 2,3,4,6,9,13,19,28,42,63,94
Otro ejemplo, desvanezca un LED hacia arriba y hacia abajo con uno `for` loop:

```
void loop() {
  int x = 1;
  for (int i = 0; i > -1; i = i + x) {
    analogWrite(PWMpin, i);
    if (i == 255) {
      x = -1; // scambiar de dirección en el pico
    }
    delay(10);
  }
}
```

4.1.8. goto

Descripción:

Transfiere el flujo del programa a un punto etiquetado en el programa.

Sintaxis:

```
label:
goto label; // envía el flujo del programa a la etiqueta
```

Código de ejemplo:

```
for (byte r = 0; r < 255; r++) {
  for (byte g = 255; g > 0; g--) {
    for (byte b = 0; b < 255; b++) {
      if (analogRead(0) > 250) {
        goto stop; //goto “lo que sea”
      }
      // más declaraciones …
    }
  }
}
stop:  // instrucción “lo que sea”:
// más declaraciones …
```

Notas y advertencias:

Se desaconseja el uso de `goto` en la programación en C, y algunos autores de libros de programación en C afirman que la instrucción `goto` nunca es necesaria, pero si se usa con prudencia, puede simplificar ciertos programas. La razón por la que muchos programadores desaprueban el uso de `goto` es que con el uso desenfrenado de sentencias `goto`, es fácil crear un programa con un flujo de programa indefinido, que nunca se puede depurar.

Dicho esto, hay casos en los que una instrucción `goto` puede ser útil y simplificar la codificación. Una de estas situaciones es salir de bucles `for` profundamente anidados, o si la lógica `if` se bloquea, en una determinada condición.

4.1.9. continue

Descripción:

La declaración `continue` salta el resto de la iteración actual de un loop (`for`, `while` o `do...while`). Continúa comprobando la expresión condicional del bucle y continúa con las iteraciones posteriores.

Código de ejemplo:

El siguiente código escribe el valor de 0 a 255 en el `PWMpin`, pero omite los valores en el rango de 41 a 119.

```
for (int x = 0; x <= 255; x ++) {
  if (x > 40 && x < 120) { // crear salto en valores
    continue;
  }
  analogWrite(PWMpin, x);
  delay(50);
}
```

4.1.10. break

Descripción:

`break` se usa para salir de un bucle `for`, `while` o `do… while`, sin pasar por la condición de bucle normal. También se usa para salir de una declaración de `switch case`.

Código de ejemplo:

En el código siguiente, el control sale del bucle for cuando el valor del sensor supera el umbral.

```
int limite = 40;
for (int x = 0; x < 255; x++) {
  analogWrite(PWMpin, x);
  sens = analogRead(sensorPin);
  if (sens > limite) { // rescatar en la detección del sensor
    x = 0;
    break;
  }
  delay(50);
}
```

4.2. Operadores aritméticos

4.2.1. % (resto)

Descripción:

La operación de **resto** calcula el resto cuando un número entero se divide por otro. Es útil para mantener una variable dentro de un rango particular (por ejemplo, el tamaño de una matriz). El símbolo `%` (porcentaje) se utiliza para realizar la operación de resto.

Sintaxis:

```
resto = dividendo % divisor;
```

Parámetros:

`resto`: variable. Tipos de datos permitidos: `int`, `float`, `double`.
`dividendo`: variable o constante. Tipos de datos permitidos: `int`.
`divisor`: variable **distinto de cero** o constante. Tipos de datos permitidos: `int`.

Código de ejemplo:

```
int x = 0;
x = 7 % 5; // x ahora contiene 2
x = 9 % 5; // x ahora contiene 4
x = 5 % 5; // x ahora contiene 0
x = 4 % 5; // x ahora contiene 4
x = -4 % 5; // x ahora contiene -4
x = 4 % -5; // x ahora contiene 4
/* actualizar un valor en una matriz cada vez a través de un bucle */
int values[10];
int i = 0;
void setup() {}
```

```
void loop() {
  values[i] = analogRead(0);
  i = (i + 1) % 10; // el operador restante pasa la variable
}
```

Notas y advertencias:

1. El operador de resto no funciona en flotantes.
2. Si el **primer** operando es negativo, el resultado es negativo (o cero). Por tanto, el resultado de `x % 10` no siempre estará entre 0 y 9 si `x` puede ser negativo.

4.2.2. * (multiplicación)

Descripción:

La **multiplicación** es una de las cuatro operaciones aritméticas primarias. El operador `*` (asterisco) opera en dos operandos para producir el producto.

Sintaxis:

```
producto = operando1 * operando2;
```

Parámetros:

`producto`: variable. Tipos de datos permitidos: `int, float, double, byte, short, long`.
`operando1`: variable o constante. Tipos de datos permitidos: `int, float, double, byte, short, long`.
`operando2`: variable o constante. Tipos de datos permitidos: `int, float, double, byte, short, long`.

Código de ejemplo:

```
int a = 5;
int b = 10;
int c = 0;
c = a * b; // la variable 'c' obtiene un valor de 50 después de
//ejecutar esta declaración
```

Notas y advertencias:

1. La operación de multiplicación puede desbordarse si el resultado es mayor que el que se puede almacenar en el tipo de datos.
2. Si uno de los números (operandos) es de tipo flotante o de tipo doble, se utilizará la matemática de coma flotante para el cálculo.
3. Si los operandos son de tipo float/doble y la variable que almacena el producto es un entero, entonces solo se almacena la parte entera y se pierde la parte fraccionaria del número.

```
float a = 5.5;
float b = 6.6;
int c = 0;
c = a * b; // la variable 'c' almacena un valor de 36 solo en
//oposición al producto esperado de 36.3
```

4.2.3. + (suma)

Descripción:

La **suma** es una de las cuatro operaciones aritméticas primarias. El operador + (más) opera en dos operandos para producir la suma.

Sintaxis:

```
sum = operand1 + operand2;
```

Parámetros:

sum: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
operand1: variable o constante. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
operando2: variable o constante. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
int a = 5;
int b = 10;
int c = 0;
c = a + b; // la variable 'c' obtiene un valor de 15 después
// de ejecutar esta declaración
```

Notas y advertencias:

1. La operación de suma puede desbordarse si el resultado es mayor que el que se puede almacenar en el tipo de datos (por ejemplo, agregar 1 a un número entero con el valor 32,767 da – 32,768).
2. La operación de suma puede desbordarse si el resultado es mayor que el que se almacena puede en el tipo de datos (por ejemplo, agregar 1 a un número entero con el valor 32,767 da – 32,768).
3. Si los operandos son de tipo flotante/doble y la variable que almacena la suma es un número entero, entonces solo se almacena la parte entera y se pierde la parte fraccionaria del número.

```
float a = 5.5;
float b = 6.6;
int c = 0;
c = a + b; // la variable 'c' almacena un valor de 12 solo en
//oposición a la suma esperada de 12.1
```

4.2.4. - (resta)

Descripción:

La **resta** es una de las cuatro operaciones aritméticas primarias. El operador − (menos) opera sobre dos operandos para producir la diferencia del segundo con respecto al primero.

Sintaxis:

```
difference = operand1 – operand2;
```

Parámetros:

`difference`: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand1` : variable o constante. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
`operand2`: variable o constante. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
int a = 5;
int b = 10;
int c = 0;
c = a - b; // la variable 'c' obtiene un valor de -5 después de
// ejecutar esta declaración
```

Notas y advertencias:

1. La operación de resta puede desbordarse si el resultado es menor que el que se puede almacenar en el tipo de datos (por ejemplo, restar 1 de un número entero con el valor -32 768 da 32 767).
2. Si uno de los números (operandos) es de tipo flotante o de tipo doble, se utilizará la matemática de coma flotante para el cálculo.
3. Si los operandos son de tipo flotante/doble y la variable que almacena la diferencia es un número entero, entonces solo se almacena la parte entera y se pierde la parte fraccionaria del número.

```
float a = 5.5;
float b = 6.6;
int c = 0;
c = a - b; // la variable 'c' almacena un valor de -1 solo en
//oposición a la diferencia esperada de -1.1
```

4.2.5. / (división)

Descripción:

La **división** es una de las cuatro operaciones aritméticas primarias. El operador `/` (barra inclinada) opera en dos operandos para producir el resultado.

Sintaxis:

```
result = numerator / denominator;
```

Parámetros:

`result`: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
`numerator`: variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
`denominator`: constante o variable **distinta de cero**. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
int a = 50;
int b = 10;
int c = 0;
```

```
c = a / b; // la variable 'c' obtiene un valor de 5 después de
//ejecutar esta declaración
```

Notas y advertencias:

1. Si uno de los números (operandos) es de tipo flotante o de tipo doble, se utilizará la matemática de coma flotante para el cálculo.
2. Si los operandos son de tipo flotante/doble y la variable que almacena el resultado es un número entero, entonces solo se almacena la parte entera y se pierde la parte fraccionaria del número.

```
float a = 55.5;
float b = 6.6;
int c = 0;
c = a / b; // la variable 'c' almacena un valor de 8 solo en
//oposición al resultado esperado de 8.409
```

4.2.6. = (operador de asignación)

Descripción:

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

Código de ejemplo:

```
int sensVal; // declarar una variable entera llamada sensVal
sensVal = analogRead(0); // almacene el voltaje de entrada
(digitalizado) en el pin analógico 0 en SensVal
```

Notas y advertencias:

1. La variable del lado izquierdo del operador de asignación (signo =) debe poder contener el valor almacenado en ella. Si no es lo suficientemente grande para contener un valor, el valor almacenado en la variable será incorrecto.
2. No confunda el operador de asignación [=] (signo igual único) con el operador de comparación [==] (signo igual doble), que evalúa si dos expresiones son iguales.

4.3. Operadores de comparación

4.3.1. < (menor que)

Descripción:

Compara la variable de la izquierda con el valor o la variable de la derecha del operador. Devuelve `true` cuando el operando de la izquierda es menor que el operando de la derecha. Tenga en cuenta que puede comparar variables de diferentes tipos de datos, pero eso podría generar resultados impredecibles; por lo tanto, se recomienda comparar variables del mismo tipo de datos, incluido el tipo `signed/unsigned`.

Sintaxis:

```
x < y; // es verdadero si x es menor que y y es falso si x  es igual
```

```
// o mayor que y
```

Parámetros:

x: variable. Tipos de datos permitidos: `int, float, double, byte, short, long`.
y: variables o constantes. Tipos de datos permitidos `int, float, double, byte, short, long`.

Código de ejemplo:

```
if (x < y) { // prueba si x es menor (menor) que y
  // hacer algo solo si el resultado de la comparación es verdadero
}
```

Notas y advertencias:

Los números negativos son menores que los números positivos.

4.3.2. <= (menor o igual que)

Descripción:

Compara la variable de la izquierda con el valor o la variable de la derecha del operador. Devuelve verdadero cuando el operando de la izquierda es menor o igual que el operando de la derecha. Tenga en cuenta que puede comparar variables de diferentes tipos de datos, pero eso podría generar resultados impredecibles; por lo tanto, se recomienda comparar variables del mismo tipo de datos, incluido el tipo `signed/unsigned`.

Sintaxis:

```
x <= y; // es verdadero si x es menor o igual que y y es falso
        // si x es mayor que y
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: variable o constante. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
if (x <= y) { // prueba si x es menor (menor) que o igual a y
  // hacer algo solo si el resultado de la comparación es verdadero
}
```

Notas y advertencias:

Los números negativos son más pequeños que los números positivos.

4.3.3. > (mayor que)

Descripción:

Compara la variable de la izquierda con el valor de la variable de la derecha del operador. Devuelve verdadero cuando el operando de la izquierda es mayor que el operando de la derecha. Tenga en cuenta que puede comparar variables de diferentes tipos de datos, pero eso podría generar resultados impredecibles; por lo tanto, se recomienda comparar variables del mismo tipo de datos, incluido el tipo `signed`/`unsigned`.

Sintaxis:

```
x > y; // es verdadero si x es mayor que y y es falso si x es
// igual o menor que y
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: variable o constante. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
if (x > y) { // prueba si x es mayor que y
// hace algo solo si el resultado de la comparación es verdadero
}
```

Notas y advertencias:

Los números positivos son mayores que los números negativos.

4.3.4. >= (Mayor o igual qué)

Descripción:

Compara la variable de la izquierda con el valor o la variable de la derecha del operador. Devuelve verdadero cuando el operando de la izquierda es mayor o igual que el operando de la derecha. Tenga en cuenta que puede comparar variables de diferentes tipos de datos, pero eso podría generar resultados impredecibles, por lo tanto, se recomienda comparar variables del mismo tipo de datos, incluido el tipo `signed/unsigned`.

Sintaxis:

```
x >= y; // es verdadero si x es mayor o igual que y y es falso si
// x es menor que y
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
if (x >= y) { // prueba si x es mayor que o igual a y
 // hacer algo solo si el resultado de la comparación es verdadero
}
```

Notas y advertencias:

Los números positivos son mayores que los números negativos.

4.3.5. == (igual que)

Descripción:

Compara la variable de la izquierda con el valor o la variable de la derecha del operador. Devuelve verdadero cuando los dos operandos son iguales. Tenga en cuenta que puede comparar variables de diferentes tipos de datos, pero eso podría generar resultados impredecibles; por lo tanto, se recomienda comparar variables del mismo tipo de datos, incluido el tipo `signed/unsigned`.

Sintaxis:

```
x == y; // es verdadera si x es igual a y es falsa si x no es
// igual a y
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
if (x == y) { // prueba si x es igual a y
  // hacer algo solo si el resultado de la comparación es verdadero
}
```

4.3.6. != (no es igual que)

Descripción:

Compara la variable de la izquierda con el valor o la variable de la derecha del operador. Devuelve verdadero cuando los dos operandos no son iguales. Tenga en cuenta que puede comparar variables de diferentes tipos de datos, pero eso podría generar resultados impredecibles, por lo tanto, se recomienda comparar variables del mismo tipo de datos, incluido el tipo `signed/unsigned`.

Sintaxis:

```
x != y; // es falsa si x es igual a y es verdadera si x no es igual a
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
if (x != y) { // prueba si x no es igual a y
  // hacer algo solo si el resultado de la comparación es verdadero
}
```

4.4. Operadores booleanos

4.4.1. ! (No lógico)

Descripción:

El **NOT lógico** da como resultado un `true` si el operando es `falso` y viceversa.

Código de ejemplo:

Este operador se puede usar dentro de la condición de una declaración `if`.

```
if (!x) { // si x no es verdad
  // declaraciones
}
```

Se puede utilizar para invertir el valor booleano.

```
x = !y; // el valor invertido de y se almacena en x
```

Notas y advertencias:

El bit a bit not ~ (tilde) se ve muy diferente al booleano not `!` (signo de exclamación o "bang" como dicen los programadores) pero todavía tienes que estar seguro de cuál quieres y dónde.

4.4.2. && (AND lógico)

El **AND lógico** da como resultado `true` **solo** si ambos operandos son verdaderos.

Código de ejemplo:

Este operador se puede usar dentro de la condición de una declaración `if`.

```
if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) {
  // Si AMBOS interruptores leen ALTO declaraciones
}
```

Notas y advertencias:

Asegúrese de no confundir el operador booleano AND, && (doble ampersand) con el operador AND bit a bit & (un ampersand). Son bestias completamente diferentes.

4.4.3. || (OR lógico)

Descripción:

El **OR lógico** da como resultado un `true` si cualquiera de los dos operandos es verdadero.

Código de ejemplo:

Este operador se puede usar dentro de la condición de una declaración `if`.

```
if (x > 0 || y > 0) { // si x o y es mayor que cero
  // declaraciones
}
```

Notas y advertencias:

No confundas el booleano `||` (doble pleca) con el operador OR bit a bit `|` (pleca única).

4.5. Operadores con punteros de acceso

4.5.1. & (Operador de referencia)

Descripción:

La referencia es una de las características específicamente para usar con punteros. El operador ampersand `&` se utiliza para este propósito. Si `x` es una variable, `&x` representa la dirección de la variable `x`.

Código de ejemplo:

```
int *p; // declarar un puntero a un tipo de datos int
int i = 5;
int result = 0;
p = &i; // ahora 'p' contiene la dirección de 'i'
result = *p; // 'result' obtiene el valor en la dirección señalada
// por 'p' es decir, obtiene el valor de 'i' que es 5
```

Notas y advertencias:

Los punteros son uno de los temas complicados para los principiantes en el aprendizaje de C, y es posible escribir la gran mayoría de los bocetos de Arduino sin encontrar punteros. Sin embargo, para manipular ciertas estructuras de datos, el uso de punteros puede simplificar el código, y el conocimiento de la manipulación de punteros es útil para tener en el conjunto de herramientas.

4.5.2. * (Operador de desreferencia)

Descripción:

La desreferenciación es una de las características específicamente para usar con punteros. El operador asterisco `*` se utiliza para este propósito. Si `p` es un puntero, entonces `*p` representa el valor contenido en la dirección apuntada por `p`.

Código de ejemplo:

```
int *p; // declarar un puntero a un tipo de datos int
int i = 5;
int result = 0;
p = &i; // ahora 'p' contiene la dirección de 'i'
result = *p; // 'resultado' obtiene el valor en la dirección señalada
// por 'p' es decir, obtiene el valor de 'i' que es 5
```

Notas y advertencias:

Los punteros son uno de los temas complicados para los principiantes en el aprendizaje de C, y es posible escribir la gran mayoría de los bocetos de Arduino sin encontrar punteros. Sin embargo, para manipular ciertas estructuras de datos, el uso de punteros puede simplificar el código, y el conocimiento de la manipulación de punteros es útil para tener en el conjunto de herramientas.

4.6. Operadores bit a bit

4.6.1. & (bit a bit and)

Descripción:

El operador AND bit a bit en C++ es un solo ampersand `&`, que se usa entre otras dos expresiones enteras. AND bit a bit opera en cada posición de bit de las expresiones circundantes de forma independiente, de acuerdo con esta regla: si ambos bits de entrada son 1, la salida resultante es 1; de lo contrario, la salida es 0.

Otra forma de expresar esto es:

```
0 0 1 1 operando1
0 1 0 1 operando2
----------
0 0 0 1 (operando1 & operando2) - resultado devuelto
```

En Arduino, el tipo int es un valor de 16 bits, por lo que usar & entre dos expresiones int hace que se produzcan 16 operaciones AND simultáneas.

Código de ejemplo:

En un fragmento de código como:

```
int a = 92; // en binario: 0000000001011100
int b = 101; // en binario: 0000000001100101
int c = a & b; // resultado: 0000000001000100, o 68 en decimal.
```

Cada uno de los 16 bits en ayb se procesa mediante AND bit a bit, y los 16 bits resultantes se almacenan en c, lo que da como resultado el valor 01000100 en binario, que es 68 en decimal.

Uno de los usos más comunes de AND bit a bit es seleccionar un bit (o bits) en particular de un valor entero, a menudo llamado enmascaramiento. Vea a continuación un ejemplo (específico de la arquitectura AVR).

```
PORTD = PORTD & 0b00000011; // borrar bits 2 - 7, dejar pins PD0 and
// PD1 intacto (xx & 11 == xx)
```

4.6.2. < < (Cambio de bit izquierdo)

Descripción:

El operador de desplazamiento a la izquierda << hace que los bits del operando izquierdo se desplacen a la **izquierda** el número de posiciones especificado por el operando derecho.

Sintaxis:

```
variable << number_of_bits;
```

Parámetros:

`variable`: Tipos de datos permitidos: `byte`, `int`, `long`.
`number_of_bits`: un número que es < = 32. Tipos de datos permitidos: `int`.

Código de ejemplo:

```
int a = 5; // binario: 0000000000000101
int b = a << 3; // binario: 0000000000101000, o 40 en decimal
```

Notas y advertencias:

Cuando cambia un valor x por y bits (x << y), los y bits más a la izquierda en x se pierden, literalmente desaparecen:

```
int x = 5; // binario: 0000000000000101
int y = 14;
int result = x << y; // binario: 0100000000000000 - El primer 1 de
// 101 fue descartado.
```

Si está seguro de que ninguno de los valores en un valor se está desplazando hacia el olvido, una forma simple de pensar en el operador de desplazamiento a la izquierda es que multiplica el operando izquierdo por 2 elevado a la potencia del operando derecho. Por ejemplo, para generar potencias de 2, se pueden emplear las siguientes expresiones:

```
Operación  Resultado
---------  ------
1 << 0       1
1 << 1       2
1 << 2       4
1 << 3       8
...
1 << 8     256
1 << 9     512
1 << 10   1024
...
```

El siguiente ejemplo se puede usar para imprimir el valor de un byte recibido en el monitor serial, usando el operador de desplazamiento a la izquierda para mover el byte de abajo (LSB) a arriba (MSB), e imprimir su valor binario:

```
// Imprime el valor binario (1 o 0) del byte
void printOut1(int c) {
  for (int bits = 7; bits > -1; bits--) {
  // Comparar bits 7-0 en byte
    if (c & (1 << bits)) {
```

```
      Serial.print("1");
    }
    else {
      Serial.print("0");
    }
  }
}
```

4.6.3. >> (Cambio de bit derecho)

Descripción:

El operador de desplazamiento a la derecha >> hace que los bits del operando izquierdo se desplacen a la **derecha** el número de posiciones especificado por el operando derecho.

Sintaxis:

```
variable >> number_of_bits;
```

Parámetros:

`variable`: Tipos de datos permitidos: `byte`, `int`, `long`.
`number_of_bits`: un número que es < = 32. Tipos de datos permitidos: `int`.

Código de ejemplo:

```
int a = 40; // binario: 0000000000101000
int b = a >> 3; // binario: 0000000000000101, o 5 en decimal
```

Notas y advertencias:

Cuando desplaza x a la derecha por y bits (x >> y), y el bit más alto en x es un 1, el comportamiento depende del tipo de datos exacto de x. Si x es de tipo int, el bit más alto es el bit de signo, que determina si x es negativo o no, como hemos comentado anteriormente. En ese caso, el bit de signo se copia en bits inferiores, por razones históricas esotéricas:

```
int x = -16; // binario: 1111111111110000
int y = 3;
int result = x >> y; // binario: 1111111111111110
```

Este comportamiento, llamado extensión de signo, a menudo no es el comportamiento que desea. En su lugar, es posible que desee que los ceros se desplacen desde la izquierda. Resulta que las reglas de desplazamiento a la derecha son diferentes para las expresiones int sin firmar, por lo que puede usar una conversión de tipo para suprimir las que se copian desde la izquierda:

```
int x = -16; // binario: 1111111111110000
int y = 3;
int result = (unsigned int)x >> y; // binario: 0001111111111110
```

Si tiene cuidado de evitar la extensión del signo, puede usar el operador de desplazamiento a la derecha >> como una forma de dividir entre potencias de 2. Por ejemplo:

```
int x = 1000;
int y = x >> 3; // división entera de 1000 por 8, causando y = 125.
```

4.6.4. ^ (bit a bit xor)

Descripción:

Hay un operador algo inusual en C++ llamado OR EXCLUSIVO bit a bit, también conocido como XOR bit a bit. (En inglés, esto generalmente se pronuncia "eks-or".) El operador XOR bit a bit se escribe con el símbolo de intercalación ^. Una operación XOR bit a bit da como resultado un 1 solo si los bits de entrada son diferentes; de lo contrario, da como resultado un 0.

Precisamente,

```
0 0 1 1 operand1
0 1 0 1 operand2
----------
0 1 1 0 (operand1 ^ operand2) - returned result
```

Código de ejemplo:

```
int x = 12; // binario: 1100
int y = 10; // binario: 1010
int z = x ^ y; // binario: 0110, o decimal 6
```

El operador ^ a menudo se usa para alternar (es decir, cambiar de 0 a 1, o de 1 a 0) algunos de los bits en una expresión entera. En una operación XOR bit a bit, si hay un 1 en el bit de máscara, ese bit se invierte; si hay un 0, el bit no se invierte y permanece igual.

```
// Nota: este código utiliza registros específicos para
//microcontroladores AVR (Uno, Nano, Leonardo, Mega, etc.)
// no compilará para otras arquitecturas
void setup() {
  DDRB = DDRB | 0b00100000; // establezca PB5 (pin 13 en Uno/Nano,
  //pin 9 en Leonardo/Micro, pin 11 en Mega) como SALIDA
  Serial.begin(9600);
}

void loop() {
  PORTB = PORTB ^ 0b00100000; // invertir PB5, dejar a otros intactos
  delay(100);
}
```

4.6.5. | (bit a bit OR)

Descripción:

El operador OR bit a bit en C++ es el símbolo de barra vertical, `|`. Como el operador `&`, `|` opera independientemente cada bit en sus dos expresiones enteras circundantes, pero lo que hace es diferente (por supuesto). El OR bit a bit de dos bits es 1 si uno o ambos bits de entrada son 1; de lo contrario, es 0.

En otras palabras:

```
0 0 1 1 operand1
0 1 0 1 operand2
----------
0 1 1 1 (operand1 | operand2) - resultado devuelto
```

Código de ejemplo:

```
int a = 92; // en binario: 0000000001011100
int b = 101; // en binario: 0000000001100101
int c = a | b; // resultado: 0000000001111101, o 125 en decimal
```

Uno de los usos más comunes de bit a bit OR es establecer múltiples bits en un número empaquetado en bits.

```
// Nota: Este código es específico de la arquitectura AVR
/* configure los bits de dirección para los pines 2 a 7, deje PD0 y
PD1 intactos (xx | 00 == xx) */
// igual que pinMode(pin, OUTPUT) para pines 2 a 7 en Uno o Nano
DDRD = DDRD | 0b11111100;
```

4.6.6. ~ (bit a bit not)

Descripción:

El operador NOT bit a bit en C++ es el carácter de tilde `~`. A diferencia de `&` y `|`, el operador NOT bit a bit se aplica a un solo operando a su derecha. Bitwise NOT cambia cada bit a su opuesto: 0 se convierte en 1 y 1 se convierte en 0.

En otras palabras:

```
0 1 operand1
-----
1 0 ~operand1
```

Código de ejemplo:

```
int a = 103; // binario: 0000000001100111
int b = ~a; // binario: 1111111110011000 = -104
```

Notas y advertencias:

Es posible que se sorprenda al ver un número negativo como -104 como resultado de esta operación. Esto se debe a que el bit más alto en una variable int es el llamado bit de signo. Si el bit más alto es 1, el número se interpreta como negativo. Esta codificación de números positivos y negativos se conoce como complemento a dos. Para obtener más información, consulte el artículo de Wikipedia sobre el complemento a dos.

Aparte, es interesante notar que para cualquier entero x , ~x es lo mismo que -x - 1 .

A veces, el bit de signo en una expresión de entero con signo puede causar algunas sorpresas no deseadas.

4.7. Operadores compuestos

4.7.1. %= (resto compuesto)

Descripción:

Esta es una forma abreviada conveniente para calcular el resto cuando un número entero se divide por otro y asignarlo de nuevo a la variable en la que se realizó el cálculo.

Sintaxis:

```
x %= divisor; // equivalente a la expresión x = x % divisor;
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`.
`Divisor`: variable o constante **distinta de cero**. Tipos de datos permitidos: `int`.

Código de ejemplo:

```
int x = 7;
x %= 5; // x ahora contiene 2
```

Notas y advertencias:

1. El operador de resto compuesto no funciona en flotantes.
2. Si el **primer** operando es negativo, el resultado es negativo (o cero). Por tanto, el resultado de `x% = 10` no siempre estará entre 0 y 9 si X puede ser negativo.

4.7.2. &= (bit a bit and compuesto)

Descripción:

El operador AND bit a bit compuesto `&=` se usa a menudo con una variable y una constante para forzar bits particulares en una variable al estado BAJO (a 0). Esto a menudo se denomina en las guías de programación como bits de "borrado" o "restablecimiento".

Una revisión del operador Bitwise AND &:

```
0 0 1 1 operando1
0 1 0 1 operando2
----------
0 0 0 1 (operando1 & operando2) – devuelve el resultado
```

Sintaxis:

```
x &= y; // equivale a x = x & y;
```

Parámetros:

x: variable. Tipos de datos permitidos: `char`, `int`, `long`.
y: variables o constantes. Tipos de datos permitidos: `char`, `int`, `long`.

Código de ejemplo:

Los bits que son "AND bit a bit" con 0 se borran a 0, por lo que, si myByte es una variable de byte,

```
myByte & 0b00000000 = 0;
```

Los bits que tienen un "AND bit a bit" con 1 no se modifican, por lo que,

```
myByte & 0b11111111 = myByte;
```

Notas y advertencias:

Debido a que estamos tratando con bits en un operador bit a bit, es conveniente usar el formateador binario con constantes. Los números siguen teniendo el mismo valor en otras representaciones, solo que no son tan fáciles de entender. Además, se muestra 0b00000000 para mayor claridad, pero cero en cualquier formato de número es cero (hmmm, ¿hay algo filosófico ahí?)

En consecuencia, para borrar (poner a cero) los bits 0 y 1 de una variable, dejando el resto de la variable sin cambios, use el operador AND bit a bit compuesto (&=) con la constante 0b11111100

```
1   0   1   0   1   0   1   0   variable
1   1   1   1   1   1   0   0   máscara
----------------------
1   0   1   0   1   0   0   0

bits sin cambios
                    bits borrados
```

Aquí está la misma representación con los bits de la variable reemplazados con el símbolo x

```
x   x   x   x   x   x   x   x   variable
1   1   1   1   1   1   0   0   máscara
----------------------
x   x   x   x   x   x   0   0

bits sin cambios
                        bits borrados
```

Así que si:

```
myByte = 0b10101010;
myByte &= 0b11111100; // da como resultado 0b10101000
```

4.7.3. *= (multiplicación compuesta)

Descripción:

Esta es una abreviatura conveniente para realizar la multiplicación de una variable con otra constante o variable.

Sintaxis:

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

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y:variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
x = 2;
x *= 2; // x ahora contiene 4
```

4.7.4. ++ (incremento)

Descripción:

Incrementa el valor de una variable en 1.

Sintaxis:

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

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `long` (posiblemente unsigned).

Devuelve:

El valor original o recién incrementado de la variable.

Código de ejemplo:

```
x = 2;
y = ++x; // x ahora contiene 3, y contiene 3
y = x++; // x contiene 4, pero y todavía contiene 3
```

4.7.5. += (suma compuesta)

Descripción:

Esta es una abreviatura conveniente para realizar sumas en una variable con otra constante o variable.

Sintaxis:

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

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
x = 2;
x += 4; // x ahora contiene 6
```

4.7.6. -- (decremento)

Descripción:

Decrementa el valor de una variable en 1.

Sintaxis:

```
x--; // decrementa x en uno y devuelve el valor anterior de x
--x; // decrementa x en uno y devuelve el nuevo valor de x
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `long` (posiblemente `unsigned`).

Devuelve:

El valor original o nuevamente decrementado de la variable.

Código de ejemplo:

```
x = 2;
y = --x; // x ahora contiene 1, y contiene 1
y = x--; // x contiene 0, pero y todavía contiene 1
```

4.7.7. -= (resta compuesta)

Descripción:

Esta es una abreviatura conveniente para realizar la resta de una constante o una variable de una variable.

Sintaxis:

```
x -= y; // equivalente a la expresión x = x – y;
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y:variables o constantes. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
x = 20;
x -= 2; // x ahora contiene 18
```

4.7.8. /= (división compuesta)

Descripción:

Esta es una abreviatura conveniente para realizar la división de una variable con otra constante o variable.

Sintaxis:

```
x /= y; // equivalente a la expresión x = x / y;
```

Parámetros:

x: variable. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.
y: constante o variable distinta de cero. Tipos de datos permitidos: `int`, `float`, `double`, `byte`, `short`, `long`.

Código de ejemplo:

```
x = 2;
x /= 2; // x ahora contiene 1
```

4.7.9. ^= (bit a bit XOR compuesto)

Descripción:

El operador XOR bit a bit compuesto ^= se usa a menudo con una variable y una constante para alternar (invertir) bits particulares en una variable.

Una revisión del operador bit a bit XOR ^:

```
0 0 1 1 operand1
0 1 0 1 operand2
----------
0 1 1 0 (operand1 ^ operand2) - devuelve result
```

Sintaxis:

```
x ^= y; // equivalente a x = x ^ y;
```

Parámetros:

x: variable. Tipos de datos permitidos: `char`, `int`, `long`.
y: variables o constantes. Tipos de datos permitidos: `char`, `int`, `long`.

Código de ejemplo:

Los bits que son "XOR bit a bit" con 0 no se modifican. Entonces, si myByte es una variable de byte,

```
myByte ^ 0b00000000 = myByte;
```

Los bits que son "XOR bit a bit" con 1 se alternan de modo que:

```
myByte ^ 0b11111111 = ~myByte;
```

Notas y advertencias:

Debido a que estamos tratando con bits en un operador bit a bit, es conveniente usar el formateador binario con constantes. Los números siguen teniendo el mismo valor en otras representaciones, solo que no son tan fáciles de entender. Además, se muestra 0b00000000 para mayor claridad, pero cero en cualquier formato de número es cero.

En consecuencia, para alternar los bits 0 y 1 de una variable, dejando el resto de la variable sin cambios, use el operador XOR bit a bit compuesto (^=) con la constante 0b00000011

```
1  0  1  0  1  0  1  0  variable
0  0  0  0  0  0  1  1  mask
----------------------
1  0  1  0  1  0  0  1

bits sin cambios
                  bits alternados
```

Aquí está la misma representación con los bits de variables reemplazados con el símbolo x. ~x representa el complemento de x.

```
x  x  x  x  x  x  x  x  variable
0  0  0  0  0  0  1  1  mask
----------------------
x  x  x  x  x  x  ~x  ~x

bits sin cambios
                  conjunto de bits
```

Así que si:

```
myByte = 0b10101010;
myByte ^= 0b00000011 == 0b10101001;
```

4.7.10. |= (bit a bit OR compuesto)

Descripción:

El operador OR bit a bit compuesto `|=` se usa a menudo con una variable y una constante para "establecer" (establecer en 1) bits particulares en una variable.

Una revisión de Bitwise OR `|` operador:

```
0   0   1   1 operando1
0   1   0   1 operando2
--------------
0   1   1   1 (operando1 | operando2) - resultado devuelto
```

Sintaxis:

```
x |= y; // equivale a x = x | y;
```

Parámetros:

x: variable. Tipos de datos permitidos: `char`, `int`, `long`.
y: variables o constantes. Tipos de datos permitidos: `char`, `int`, `long`.

Código de ejemplo:

Los bits que son "OR bit a bit" con 0 no se modifican, por lo que si myByte es una variable de byte,

```
myByte | 0b00000000 = myByte;
```

Los bits que son "OR bit a bit" con 1 se establecen en 1, por lo que:

```
myByte | 0b11111111 = 0b11111111;
```

Notas y advertencias:

Debido a que estamos tratando con bits en un operador bit a bit, es conveniente usar el formateador binario con constantes. Los números siguen teniendo el mismo valor en otras representaciones, solo que no son tan fáciles de entender. Además, se muestra 0b00000000 para mayor claridad, pero cero en cualquier formato de número es cero.

En consecuencia, para establecer los bits 0 y 1 de una variable, mientras deja el resto de la variable sin cambios, use el operador OR bit a bit compuesto (|=) con la constante 0b00000011

```
1   0   1   0   1   0   1   0   variable
0   0   0   0   0   0   1   1   mask
-------------------------------
1   0   1   0   1   0   1   1

bits sin cambios
                        conjunto de bits
```

Aquí está la misma representación con los bits de variables reemplazados con el símbolo x

```
x   x   x   x   x   x   x   x   variable
0   0   0   0   0   0   1   1   mask
-------------------------------
x   x   x   x   x   x   1   1

bits sin cambios
                       conjunto de bits
```

Así que si:

```
myByte = 0b10101010;
myByte |= 0b00000011 == 0b10101011;
```