El Lenguaje de Martuino

El lenguaje C

El lenguaje C es uno de los más rápidos y potentes que hay hoy en día. Algunos dicen que está desfasado (ja, ja) y que el futuro es Java. No sé si tendrá futuro pero está claro que presente si tiene. No hay más que decir que el sistema operativo Linux está desarrollado en C en su práctica totalidad. Así que creo que no sólo no perdemos nada aprendiéndolo sino que ganamos mucho. Para empezar nos servirá como base para aprender C++ e introducirnos en el mundo de la programación Windows. En el mercado hay muchas tarjetas que utilizan este lenguaje, incluida Martuino.

No debemos confundir C con C++, que no son lo mismo. Se podría decir que C++ es una extensión de C. Para empezar en C++ conviene tener una sólida base de C.

Existen otros lenguajes como Visual Basic que son muy sencillos de aprender y de utilizar. Nos dan casi todo hecho. Pero cuando queremos hacer algo complicado o que sea rápido debemos recurrir a otros lenguajes (c++, delphi,...).

Una de las cosas importantes de C que debes recordar es que es *Case Sensitive* (sensible a las mayúsculas o algo así). Es decir que para C no es lo mismo escribir *Printf* que *printf*.

Conviene indicar también que las instrucciones se separan por ";".

Un compilador es un programa que convierte nuestro código fuente en un programa ejecutable (Me imagino que la mayoría ya lo sabéis, pero más vale asegurar). El ordenador trabaja con 0 y 1. Si escribiéramos un programa en el lenguaje del ordenador nos volveríamos locos. Para eso están lenguajes como el C. Nos permiten escribir un programa de manera que sea fácil entenderlo por una persona (el **código fuente**). Luego es el compilador el que se encarga de convertirlo al complicado idioma de Martuino.

En la práctica a la hora de crear un programa nosotros escribimos el código fuente, en nuestro caso en C, que normalmente será un fichero de texto normal y corriente que contiene las instrucciones de nuestro programa. Luego se lo pasamos al compilador y este se encarga de convertirlo en un programa.

El Editor para que escribamos un programa en el lenguaje C y el compilador lo traduzca al lenguaje de Martuino, es el programa Energia, CCS de Texas intruments o incluso Visual Studio community 2015/2017/2019 de Microsoft. En todos estos programas puedes crear el tuyo y enviarlo a la placa Martuino. Si no estás familiarizado con

ninguno de estos programas, te aconsejamos que utilices Visual studio community 2015/2017/2019 de Microsoft y un plugin que podrás descargarte en la página web http://martuino.com/software.html. Visual studio es totalmente gratuito y cuando escribes un comando de los que a continuación te explicamos, el solo te muestra el resto del comando, así como todas sus posibilidades, ayudante a escribir tus programas. No te asustes si te parece muy complicado, no lo es y además tenemos una guía que te dice como instalarlo y ponerte a programar.

Como hemos comentado, cuando escribes un programa en C las líneas de código acaban todas con ";" excepto los comentarios. También cuando creas una parte del programa llamado función, se comienza con el símbolo llave "{"y finaliza con "}".

Cualquier programa de Martuino debe de tener dos partes fundamentales y necesarias, se pueden llamar funciones principales:

La primera será:

void setup()

{

}

Donde pondremos el inicio de nuestro programa, por ejemplo, pondremos los pines de la placa que son entradas y cuales salidas, la velocidad del puerto serie, el inicio del wifi o del i2c, etc. Se pueden añadir variables o constantes, ya verás luego que son todos estos términos. También se pueden declarar variables antes del "void setup()".

La segunda será:

void loop()

{

}

Como veras "loop" significa bucle, es decir, que el programa contenido dentro de las llaves "{ }" se ejecutará infinitamente, excepto si le ponemos una instrucción de parar o salir.

A estas dos funciones principales se le puede añadir cualquier función que necesitemos en nuestro programa, incluido las interrupciones.

Hemos puesto las dos funciones principales que necesita cualquier programa para Martuino, ahora veremos las palabras que se utilizan en forma de comando, y solo se pueden utilizar como comandos y no como variables, en orden alfabético. No es necesario sabérselas, siempre podrás echarle un vistazo a este manual cuando no sepas como realizar alguna operación con el programa.

Estos son los comandos en C para programar Martuino que puedes utilizar. Los nombres de estos comandos no se pueden utilizar como nombres de variables, o etiquetas.

Comandos por orden alfabético:

abs(x)

Descripción
Computa el valor absoluto de un número.

Parámetros
x: el número

Retorna
x: si **x** es mayor o igual que 0.

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

analogRead()

Descripción
Lee el valor de un determinado pin analógico. Martuino contiene 3 canales de 12 bits que convierten de analógico a digital. La medida de cada entrada tendrá como valor máximo 1,45 voltios que corresponderá al valor 4095 en decimal. Por lo tanto la resolución de la lectura será de: 1,45/4096 = 0.354mv por bit. El rango de la resolución puede cambiarse usando el comando analogReference().

Realizar una medida toma 100 microsegundos (0.0001 s) , esto implica que la máxima velocidad de lectura será de 10.000 veces por segundo.

Cuando realices una medida analógica, y quieras saber la tensión que has medido del pin utiliza esto:

```
float valor_real = 0;

int sensorvalue = 0;

sensorvalue = analogRead(16);
valor_real = (sensorvalue * (1.45 / 4095));
```

Sintaxis

analogRead(pin)

Parámetros

pin: número de la entrada analógica a leer (puede ser 16, 17 y 18).

Retorna

int (0 a 4095)

analogWrite()

Descripción

Escribe un valor analógico (PWM) a un pin. Puede ser usado para variar la luz de un LED o variar la velocidad de un motor. Después de una llamada a **analogWrite()**, el pin generará una onda cuadrada con un especificado ciclo hasta la próxima llamada a **analogWrite()** (o a **digitalRead()** o **digitalWrite()** en el mismo pin). La frecuencia de la señal PWM es de aproximadamente 490 Hz.

La función *analogWrite* no tiene nada que ver con un pin analógico o la función *analogRead* .

Sintaxis

analogWrite(pin, valor)

Parámetros

pin: el pin.

Valor: ciclo de funcionamiento entre 0 y 255 (siempre “0” y siempre “1”).

Arrays

Un array es una colección de variables que pueden ser accesibles con un índice numérico. Arrays en el lenguaje de programación C , puede resultar complicado, pero usándolo en modo simple es relativamente sencillo.

Creando (Declarando) un Array

Todos estos métodos mostrados son válidos para crear (declarar) un array.

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

Tú puedes declarar un array sin inicializarlo como en myInts.

En myPins declaramos un array sin un tamaño explícito. El compilador contará los elementos y creará un array de tamaño apropiado.

Finalmente puedes crear un array inicializando su tamaño y sus elementos, como en mySensVals. Nota, cuando declares un array de tipo char, más de un elemento es requerido, o en su defecto el carácter null.

Accediendo al Array

El primer elemento de un array es el elemento cero (0).
`mySensVals[0] == 2, mySensVals[1] == 4`, etc.

También significa que un array con 10 elementos, tendrá como último elemento el 9.

Por lo tanto:

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

Por esta razón hay que ser cuidadoso accediendo a los arrays. Si cuando accedes a un array te pasas el final de este (usando un índice más grande que el declarado como tamaño del array -1) estás leyendo de la memoria que está en uso para otros propósitos. Leyendo esta otra localización es probable que vayas a un dato inválido. Escribir en una posición de la memoria aleatoriamente es mala idea y puede ocasionar que el programa se cuelgue o funcione mal. Esto puede ser difícil de detectar y volverte loco buscando un fallo en el código.

Diferente a BASIC o JAVA, el compilador C no chequea si se accede a un elemento mayor que el tamaño de un array, aunque haya sido declarado su tamaño.

Para asignar un valor a un array:

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

Para leer un valor desde un array:

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

attachInterrupt(interrupción, función, modo)

Descripción

Especifica la función que será llamada cuando ocurra una interrupción externa.

Parámetros

interrupción: el número de interrupción (*int*)

función: la función a la que llamará cuando ocurra una interrupción; Esta función no debe de tomar ningún parámetro y no devolverá nada (return). Esta función es a veces llamada como rutina de interrupción(*interrupt service routine).*

modo define como la interrupción será disparada. Cuatro constantes son predefinidas:

LOW disparará la interrupción cuando el pin este a nivel bajo,
CHANGE disparará la interrupción cuando el valor del pin cambie,
RISING disparará cuando el pin este a nivel alto,
FALLING disparará la interrupción cuando el pin pase de nivel alto a bajo.

Retorna

nada

Nota

Dentro de la función de interrupción , delay() no trabaja y el valor retornado en milisegundos (millis()) no será incrementada. Los datos del puerto serie serán perdidos. No podrás declarar como volatile ninguna variable que tu modifiques dentro de la función de la interrupción.

Usando las Interrupciones

Las interrupciones son utilizadas para hacer cosas automáticamente en los programas de los microcontroladores, y pueden ayudar a resolver algunos problemas. Una buena tarea es usar una interrupciones para leer un encoder y monitorizando una entrada.

Si quieres asegurarte que un programa siempre coja el pulso de un encoder rotary, que nunca pierda un pulso, es difícil de realizar con un programa normal, porque el programa necesitaría estar constantemente leyendo la entrada del encoder, esperando a que ocurra un pulso. También hay otros sensores, como un sensor de audio o de infrarrojos, que también resultan difíciles de leer. En todas estas situaciones es mejor utilizar interrupciones para poder dejar libre al microprocesador haciendo otros trabajos mientras no perderá ninguna otra señal.

Ejemplo

```
volatile int state = HIGH;
volatile int flag = HIGH;
int count = 0;
```

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

  pinMode(GREEN_LED, OUTPUT);
  digitalWrite(GREEN_LED, state);
  attachInterrupt(PUSH2, blink, FALLING); // Interrupción ocurre
cuando el botón es presionado
}

void loop()
{
  digitalWrite(GREEN_LED, state); //LED en marcha
  if(flag) {
    count++;
    Serial.println(count);
    flag = LOW;
  }
}

void blink()
{
  state = !state;
  flag = HIGH;
}
```

bit()

Descripción

Computa el valor de un especificado bit (bit 0 es el 1, bit 1 es el 2, bit 2 es el 4, etc.).

Sintaxis

bit(n)

Parámetro

n: el bit a valorar (0-7), siendo el 0 el menos significativo, el de la derecha.

Retorna

El valor de ese bit

bitClear()

Descripción

Limpia (escríbelo a 0) un bit de una variable numérica.

Sintaxis
bitClear(x, n)

Parámetros
x: la variable numérica en la que quieres limpiar un bit

n: el bit a limpiar, comenzando por 0 para el bit menos significativo (derecho) bit (0-7)

Retorna
nada

bitRead()

Descripción
Lee un bit de un número.

Sintaxis
bitRead(x, n)

Parámetros
x: el número desde donde se leerá

n: el bit a leer, comenzando por 0 para el bit menos significativo (derecho) bit (0-7)

Retorna
El valor del bit (0 o 1).

bitSet()

Descripción
Escribe a "1" un bit de una variable numérica.

Sintaxis
bitSet(x, n)

Parámetros
x: La variable numérica donde quieres poner el bit a "1"

n: bit puesto a 1, comenzando por el menos significativo(el situado más a la derecha)

Retorna
nada

bitWrite()

Descripción
Escribe un bit de una variable numérica.

Sintaxis
bitWrite(x, n, b)

Parámetros
x: la variable numérica que será escrita

n: el número de bit a escribir, comenzando por el menos significativo(el situado más a la derecha)

bel valor a escribir en el bit (0 o 1)

Retorna
Nada

boolean

Un **boolean** solo puede tomar dos valores, true o false(verdadero o falso). (Cada variable boolean ocupa un byte de memoria.)

Ejemplo
```
int LEDpin = GREEN_LED;        // LED en pin 14
int switchPin = PUSH2;    // pin 5

boolean running = false;

void setup()
{
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);    // activa la resistencia interna
a vcc
}
```

```
void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // Pulsación - mantiene el pin a alto
    delay(200);                         // espera que pasen los rebotes
    running = !running;                 // invierte la variable
    digitalWrite(LEDpin, running)       // indícalo vía LED
  }
}
```

break

break es usado para salir de una sentencia **do**, **for**, o bucle **while** , sobrepasando la condición del bucle. Este comando también es usado para salir de una sentencia **switch**.

Ejemplo

```
for (x = 0; x < 255; x ++)
{
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      // salte cuando detecte el sensor
       x = 0;
       break;
    }
    delay(50);
}
```

byte

Descripción

Un byte es un tipo de dato que almacena 8-bit sin signo, es decir, desde 0 hasta 255.

Ejemplo

```
    byte b = B10010;  // "B" tiene formato binario (B10010 = 18
decimal)
```

byte()

Descripción
Convierte un valor a tipo byte.

Sintaxis
byte(x)

Parámetros
x: un valor de cualquier tipo

Retorna
byte

Caracteres ASCII

El código ASCII (American Standard Code for Information Interchange) data de los años 1960's. Es un estándar de texto codificado numéricamente.

Nota: los primeros 32 caracteres (0-31) son caracteres no imprimibles, son llamados caracteres de control. Los caracteres más utilizados han sido etiquetados.

DEC Valor	Carácter	DEC Valor	Carácter	DEC Valor	Carácter	DEC Valor	Carácter
0	null	32	espac	64	@	96	`
1		33	!	65	A	97	a
2		34	"	66	B	98	b
3		35	#	67	C	99	c
4		36	$	68	D	100	d
5		37	%	69	E	101	e
6		38	&	70	F	102	f
7		39	'	71	G	103	g
8		40	(	72	H	104	h
9	tab	41	)	73	I	105	i
10	salto de línea	42	*	74	J	106	j
		43	+	75	K	107	k
11		44	,	76	L	108	l
12		45	-	77	M	109	m
13	retorno de carro	46	.	78	N	110	n
		47	/	79	O	111	o
14		48	0	80	P	112	p
15		49	1	81	Q	113	q
16		50	2	82	R	114	r
17		51	3	83	S	115	s
18		52	4	84	T	116	t
19		53	5	85	U	117	u

20	54 6	86 V	118 v
21	55 7	87 W	119 w
22	56 8	88 X	120 x
23	57 9	89 Y	121 y
24	58 :	90 Z	122 z
25	59 ;	91 [	123 {
26	60 <	92 \	124 |
27	61 =	93]	125 }
28	62 >	94 ^	126 ~
29	63 ?	95 _	127
30			
31			

char

Descripción

Un tipo de dato que utiliza hasta 1 byte de memoria y almacena un valor de un solo carácter. Lo caracteres literales son escritos con cimillas simples, como : 'A' (para múltiples caracteres - strings – son usadas dobles comillas: "ABC").

Los caracteres son almacenados como números. Puedes ver el código especifico en la tabla de caracteres ASCII. Esta forma hace posible que un carácter sea un número, correspondiente en la tabla ASCII con un valor (ejemplo 'A' + 1 tiene un valor 66, corresponde el valor ASCII de la letra mayúscula A que es 65).

El tipo char no tiene signo, para no confundir con un byte con signo que podría tomar valores desde -128 to 127. Para un byte sin signo (8 bit) , usa el tipo de dato *byte*.

Ejemplo

```
char myChar = 'A';
char myChar = 65;      // ambos son equivalentes
```

char()

Descripción

Convierte una valor a un tipo char.

Sintaxis

char(x)

Parámetros

x: un valor de cualquier tipo.

Retorna

char

Comentarios

Los comentarios son líneas en un programa que se utilizan para informarse como trabaja el programa. Estas líneas son ignoradas por el compilador, y no se exportan al procesador, así que estas líneas no ocupan espacio alguno en la memoria del microcontrolador.

Los comentarios solo se utilizan como ayuda para entender (o recordar) como trabaja un programa o para informar a otros como trabaja tú programa. Hay dos diferentes maneras de hacer líneas como comentarios:

Ejemplo

```
 x = 5;  // Este es una simple línea de comentarios.
         // Esto es el final de la línea.

/* Estas son múltiples líneas de comentarios - úsalo para comentarios
en bloques

if (gwb == 0){   // simple línea de comentarios
x = 3;           /* pero no comentarios multilínea - esto es invalido
*/
}
// no olvidar cerrar los comentarios
*/
```

const

La palabra **const** está para constantes. Esta es una variable *qualifier* que modifica el comportamiento de una variable, haciéndola solo de lectura. Esto provoca que la variable pueda ser usada como cualquier otra variable de ese tipo, pero no pueda cambiarse su valor. Obtendrás un error del compilador si pruebas a asignar un valor a una variable **const** .

Una constante se define con la palabra cons y se puede utilizar *#define*, para que tome un valor concreto.

Ejemplo

```
const float pi = 3.14;
float x;

// ....
```

```
x = pi * 2;     // esto está bien.

pi = 7;         // ilegal - no puedes escribir o (modificar) una
constante
```

#define o const

Puedes usar ambas palabras **const** o **#define** para crear constantes numéricas o de tipo string. Para arrays, necesitarás usar **const**. En general *const es preferido sobre #define* para definir constantes.

Constantes

Constantes son variables predefinidas en el entorno Martuino. Ellas son utilizadas para hacer más comprensible un programa . Las clasificamos por grupos.

Definición a nivel lógico, true y false (Constantes Boolean)

Estas son dos, utilizadas para decir si es verdadero o falso : **true**, y **false**.

false

false es la más fácil d definir de las dos. false es definida como un 0 (cero).

true

true es a menudo definida como un 1, pero no es correcto, true tiene una definición más amplia. Cualquier entero que no sea cero es TRUE, en una expresión Boolean . Así -1, 2 y -200 son todas definidas como verdadero, también, en una expresión Boolean sense.

Nota: las constantes *true* y *false* son diferentes a HIGH, LOW, INPUT, y OUTPUT.

Definición a nivel de Pin, HIGH y LOW

Cuando leemos o escribimos en un pin digital solo dos posibles valores pueden ser tomados: **HIGH** y **LOW**.

HIGH

El significado de HIGH (en referencia a un pin) es algo diferente dependiendo de si un pin es una entrada o salida (INPUT o OUTPUT). Cuando un pin es configurado como entrada (INPUT) con el comando pinMode, y leído con el comando digitalRead, el microcontrolador reportará HIGH si la tensión presente en el pin es de 3 voltios o superior.

Un pin que ha sido configurado como entrada (INPUT) con el comando pinMode, y después hecho HIGH con el comando digitalWrite, este tiene una resistencia interna de

20k puesta a vcc, que dirigirá el pin de entrada a HIGH, excepto que un circuito externo lo tire a nivel bajo LOW.

Cuando un pin es configurado como salida (OUTPUT) con el comando pinMode, y puesto a nivel alto (HIGH) con el comando digitalWrite, el pin estará a 3 voltios. Este estado puede cambiar suponiendo que la corriente de carga, por ejemplo una luz de un led conectada en serie con una resistencia a masa, sea demasiado poniendo a nivel bajo LOW la salida.

LOW

El significado de LOW es algo diferente dependiendo si un pin es entrada(INPUT) o salida(OUTPUT). Cuando el pin es configurado como entrada(INPUT) con el comando pinMode, y leído con el comando digitalRead, El microcontrolador reportará un nivel bajo (LOW) si la tensión es de 2 voltios o menos en el pin seleccionado.

Cuando el pin es configurado como salida(OUTPUT) con el comando pinMode, y puesto a nivel bajo (LOW) con el comando digitalWrite, el pin estará a 0 voltios. En este estado , por ejemplo con un led conectado con una resistencia a + 3 voltios y configurada como salida, se encenderá el led.

Definición de Pin Digital , INPUT y OUTPUT

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

Pines configurados como entradas

Los pines de Martuino configurados como entrada (**INPUT) con el comando** pinMode() son pasados a estado de alta impedancia. La resistencia aproximada de los pines será de 100 Megohmios , ideal para lectura de sensores, pero no para encender un led.

Pines Configurados como salidas

Los pines configurados como salidas (**OUTPUT)** con el comando pinMode() son pasados a estado de baja impedancia. Esto significa que pueden soportar una corriente sustancial de otros circuitos. Esta corriente puede ser de hasta 40 mA (miliamperios) . Esta corriente es suficiente para encender un led directamente conectado a un pin de salida. Los pines configurados como salidas pueden ser dañados o destruidos si se cortocircuitan a masa o se les introduce 3 voltios estado en estado bajo (LOW). La corriente que puede soportar un pin de salida no es suficiente para conectar motores o relés (si sobrepasan 40mA) o cualquier circuito que requiera más de esta corriente.

Pines Configurados como entrada Pullup

Los pines de Martuino pueden ser configurados como entradas con resistencia a positivo (**INPUT_PULLUP)**. Esto hace que la entrada no se quede en un estado indeterminado, teniendo una resistencia conectada internamente a alimentación positiva. Así la lectura de este pin cuando no tenga nada conectado será de nivel alto (HIGH). Cuando se le conecte algo a una entrada de este tipo cambiará el estado de la entrada según el circuito conectado a ella, no dañando nada de dicha entrada.

```
pinMode(pin, INPUT_PULLUP);           // pin como entrad con
resistencia interna a +vcc
```

constrain(x, a, b)

Descripción

Compara un número dentro de un rango.

Parámetros

x: número a comparar, para cualquier tipo de dato.

a: rango inferior del número, para cualquier tipo de dato.

b: rango superior del número, para cualquier tipo de dato.

Retorna

x: si **x** esta entra **a** y **b**

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

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

Ejemplo

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

continue

El comando continue salta al siguiente estamento de un bucle (**do**, **for**, o **while**). Continuará testeando la expresión del bucle, y procediendo con el siguiente estamento.

Ejemplo

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

    digitalWrite(PWMpin, x); // solo se escribirán los valores que no
//están dentro de la sentencia continue
    delay(50);
}
```

cos(rad)

Descripción

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

Parámetros

rad: el ángulo en radianes (*float*)

Retorna

El coseno del ángulo ("double")

Define

`#define` es usado en C como un componente que permite al programador dar un nombre a una constante antes que el programa sea compilado. Define constantes en martuino no tomando ningún espacio en memoria del chip. El compilador reemplazará las referencias a la constante definida por el valor también definido con esta instrucción.

En general la palabra *const* es preferida para definir una constante antes que utilizar #define.

Sintaxis

```
#define Nombre constante   valor
```

Nota: es necesario# antes de define.

Ejemplo

```
#define ledPin 3
// el compilador reemplazará el nombre de ledPin con el valor 3.
```

Ojo

No debes de poner punto y coma ";" después de #define . Si lo pones el compilador dará un error.

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

Igualmente poner un = para asignar un valor a un #define también causará un error al compilar.

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

delay()

Descripción

Pausa el programa durante el tiempo especificado con parámetro (en milisegundos) . (Recuerda que 1000 milisegundos son un segundo.)

Sintaxis

delay(milisegundos)

Parámetros

número de milisegundos para la pausa(*unsigned long*)

Retorna

Nada

Ejemplo

```
int ledPin = 14;                 // LED conectado a pin 14

void setup()
{
  pinMode(ledPin, OUTPUT);      // configúralo como salida
}

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

Advertencia

Mientras es fácil crear un intermitente con un comando delay(),es también un inconveniente. Mientras está en funcionamiento el delay() no se pueden leer sensores, ni hacer cálculos o manipular pines de salida, es decir, que el microcontrolador está congelado. Como alternativa al control de retardos, échale un vistazo al comando millis() más adelante. Normalmente de utiliza delay() para tiempos menores de 10's milisegundos.

También es cierto que el comando delay() no deshabilita interrupciones, las comunicaciones serie son almacenadas y cualquier PWM (analogWrite) mantiene su valor en el pin, y las interrupciones seguirán trabajando.

delayMicroseconds()

Descripción

Pausa el programa durante un tiempo (en microsegundos) especificado en el parámetro. Recuerda que mil microsegundos son un milisegundo, y así un millón de microsegundos son un segundo.

Normalmente el mayor valor de un retardo que se produce con exactitud es 16383. Para retardos mayores de algunos miles de microsegundos, es mejor utilizar delay().

Sintaxis

delayMicrosegundos(us)

Parámetros

us: el número de microsegundos para la pausa (*unsigned int*)

Retorna

Nada

Ejemplo

```
int outPin = 14;                 // pin digital 14

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

void loop()
{
  digitalWrite(outPin, HIGH);    // enciende led
  delayMicroseconds(16383);      // pausa de 16383 microsegundos
  digitalWrite(outPin, LOW);     // apaga el led
  delayMicroseconds(16383);      // pausa de 16383 microsegundos
}
```

Advertencias y conocidas cuestiones

Este comando trabaja con exactitud en rango de más de 3 microsegundos. No aseguramos que este comando sea preciso para grandes retardos en el tiempo.

Normalmente, delayMicroseconds() no deshabilita interrupciones.

detachInterrupt(interrupción)

Descripción

Inhabilita interrupciones.

Parámetros

interrupt: el número de interrupción a deshabilitar (0 o 1).

Desplazamiento izquierdo (<<), desplazamiento derecho (>>)

Descripción

Desplazamiento a la izquierda o a la derecha de un bit.

El desplazamiento a la izquierda es el operador << y a la derecha es el operador >>. Estos operadores causan que el bit a la izquierda del operando sea desplazado a la izquierda o a la derecha tantas veces como el número especificado a la derecha del operando.

Sintaxis

variable << número de bits a desplazar a la izquierda

variable >> número de bits a desplazar a la derecha

Parámetros

variable - (byte, int, long) número de bits a desplazar <= 32

Ejemplo:

```
    int a = 5;         // binary: 0000000000000101
    int b = a << 3;    // binary: 0000000000101000, o 40 en decimal
    int c = b >> 3;    // binary: 0000000000000101, o 5 en decimal
```

Cuando desplazas a la izquierda un valor x por y bits (x << y), el más a la izquierda y bits en x son perdidos, literalmente desplazados fuera del número:

```
    int a = 5;         // binario: 0000000000000101
    int b = a << 14;   // binario: 0100000000000000 - el primer 1 en
101 será perdido.
```

Si estás seguro que ninguno de los "1" va a ser perdido , una vía simple de hacer el desplazamiento a la izquierda es la multiplicación por 2, o lo que son potencias de 2.

La siguiente expresión puede ser empleada:

```
    1 <<  0  ==    1
    1 <<  1  ==    2
    1 <<  2  ==    4
    1 <<  3  ==    8
    ...
    1 <<  8  ==  256
    1 <<  9  ==  512
    1 << 10  == 1024
    ...
```

Cuando desplazas a la derecha un valor x por y bits (x >> y), y el bit más alto en x es un "1", El comportamiento depende exactamente del tipo de x. Si x es un tipo int (entero), el bit más alto es el bit de signo, determinando sí x es negativo o no, como hemos

discutido con anterioridad. En este caso el bit significativo será copiado al bit más bajo, por razones históricas:

```
    int x = -16;      // binary: 1111111111110000
    int y = x >> 3;   // binary: 1111111111111110
```

Este comportamiento, Llamado extensión del signo, no es a menudo el comportamiento deseado. En su lugar, podemos querer que entren ceros por la izquierda. Resulta que el desplazamiento a la derecha tiene unas normas diferentes para la expresiones unsigned int(entero sin signo), Así puedes utilizar este tipo para suprimir que entren unos por la izquierda:

```
    int x = -16;                    // binary: 1111111111110000
    int y = (unsigned int)x >> 3;   // binary: 0001111111111110
```

Si tienes cuidado y evitas la extensión del signo (entrada de "1" por la izquierda), puedes usar el operador de desplazamiento a la derecha >> como una manera de dividir en potencia de 2. Por ejemplo:

```
    int x = 1000;
    int y = x >> 3;    // división entera de 1000 por 8, causando y =
125.
```

digitalRead()

Descripción

Lee el valor de un pin digital, será nivel alto (HIGH) o nivel bajo (LOW).

Sintaxis

digitalRead(pin)

Parámetros

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

Retorna

HIGH o LOW

Ejemplo

```
int val = 0;      // variable para almacenar el valor leído
void setup()
{
  pinMode(RED_LED, OUTPUT);      // RED_LED como salida
  Serial.begin(9600);
}
```

```
void loop()
{
  P1OUT = ~P1OUT;
  val = digitalRead(RED_LED); // Lee el pin de salida
  Serial.print(val);
}
```

Nota

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

Los pines analógicos pueden ser utilizados como pines digitales.

digitalWrite()

Descripción

Escribe un nivel alto (HIGH) o nivel bajo (LOW) en un pin digital.

Si el pin se ha configurado como salida (OUTPUT) con el comando pinMode(), la tensión correspondiente a cada nivel será: 3V para HIGH, 0V (masa) para LOW.

Si el pin está configurado como entrada (INPUT), escribiendo un valor HIGH con el comando digitalWrite() habilitará la resistencia interna conectada a +vcc. Escribiendo LOW se deshabilitará la resistencia a +vcc interna.

Sintaxis

digitalWrite(pin, valor)

Parámetros

pin: el número del pin

valor: HIGH o LOW

Retorna

Nada

Ejemplo

```
int ledPin = 14;                  // ledpin en pin 14

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

```
void loop()
{
  digitalWrite(ledPin, HIGH);   // enciende led
  delay(1000);                  // retardo 1 segundo
  digitalWrite(ledPin, LOW);    // apaga led
  delay(1000);                  // retardo 1 segundo
}
```

Nota

Los pines analógicos pueden ser usados como pines digitales.

double

Descripción

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

La implementación de double es exactamente igual a la del tipo float, no ganando nada en precisión.

do - while

El bucle **do** de la misma manera que el bucle **while** , con la excepción que las condiciones son testeadas al final del bucle, así el bucle **do** siempre se ejecuta una vez al menos.

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

Ejemplo

```
do
{
  delay(50);          // espera a que el sensor se estabilice
  x = readSensors();  // testea el sensor

} while (x < 100);
```

if / else

if/else permite el control sobre bloques de código fácilmente, permitiendo testear múltiples grupos juntos. Por ejemplo, una entrada analógica es testeada y determinamos con una instrucción , si su valor es menor de 500, o si es igual o mayor de 500. Este el código del ejemplo:

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

else puede producir otro **if** , y así múltiples sentencias que pueden testear al mismo tiempo.

Cada testeo se produce hasta que un verdad es encontrada. Cuando una verdad es encontrada, el bloque de código siguiente se ejecuta, y el programa se sale del completo if/else . Si no aparece ninguna sentencia a verdad se ejecuta el bloque **else** por defecto, si existe.

Nota: en un bloque **else if** puede ser usado con o sin terminación de un bloque **else** y viceversa. Se permiten un número ilimitado de grupos **else if** .

```
if (pinFiveInput < 500)
{
  // haz esto A
}
else if (pinFiveInput >= 1000)
{
  // haz esto B
}
else
{
  // haz esto C
}
```

Otra forma de hacer lo mismo que un grupo de if/else es switch case .

float

Descripción

Tipo de datos para números de coma flotante, siendo un número que tiene una parte decimal. Los número de coma flotante son usados en valores leídos de entradas analógicas , ya que da más resolución que un entero. Los números de coma flotante no pueden ser más grandes que 3.4028235E+38 y más pequeños que -3.4028235E+38. Ellos utilizan 32 bits (4 bytes) de memoria.

Los números Floats solo tienen 6-7 decimales de precisión. Que son el total de números de dígitos, no de números a la derecha de la coma decimal. En otras plataformas con el tipo double puedes tener hasta 15 dígitos de precisión, en Martuino la precisión es la misma para el tipo float como para el double.

Los tipos Float no son exactos, es decir, que por ejemplo `6.0 / 3.0` no es igual a `2.0`. En s lugar, debes de comprobar que el valor absoluto de la diferencia entre el número es menor que un número pequeño.

Para el cálculo de operaciones con tipos Float, hay que tener en cuenta que el microcontrolador tardará más tiempo en hacer una operación, que si fueran de tipo int, número entero.

Ejemplo

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

Sintaxis

```
    float var = val;
```

var – tu nombre de la variable
val – el valor asignado a esa variable

Ejemplo

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

   x = 1;
   y = x / 2;            // y contiene 0, int no tiene decimales
   z = (float)x / 2.0;   // z contiene .5 (tienes que usar 2.0, no 2)
```

float()

Descripción

Convierte una valor a tipo float .

Sintaxis

float(x)

Parámetros

x: un valor de cualquier tipo

Retorna

float

for

Descripción

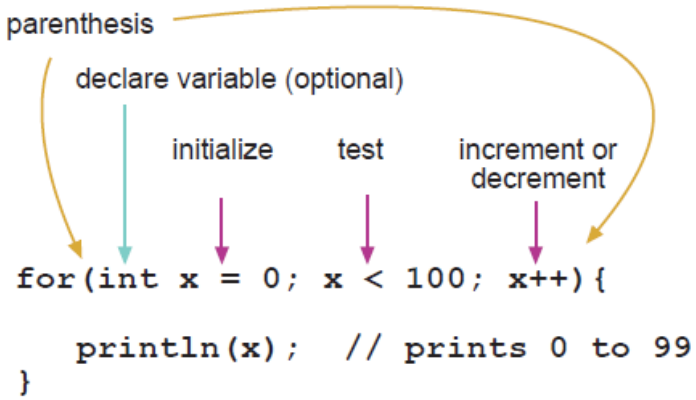
La instrucción **for** es usada para repetir bloques de código encerrados en llaves "{}". El incremento de un contador es utilizado para tener el número de repeticiones del bucle. La instrucción **for** es usada para cualquier operación repetitiva, y usada a menudo en combinación con arrays para realizar operaciones con colecciones de datos.

El bucle **for** tiene una cabecera de tres partes:

```
for (inicialización; condición; incremento) {

//código

}
```



La **inicialización** ocurre al principio y solo una vez. Cada vez que se repite el bucle , la **condición** es testeada; Si es verdad ,es decir, se cumple, se ejecuta el código, e **incrementa** se ejecuta, luego se testea de nuevo la **condición** . Cuando la **condición** se convierte en falsa, el bucle se acaba.

Ejemplo

```
// Control de un LED usando un pin PWM
int PWMpin = GREEN_LED; // Led con resistencia en pin 14
void setup()
{
  // no es necesario nada
}

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

Consejos en código

En C los bucles **for** son mucho más flexibles que otros bucles **for** en otros lenguajes, incluyendo el BASIC. Entre las tres condiciones de inicio un ";" es necesario , si no se producirá un error. Estas tres condiciones pueden ser variables de todo tipo, incluso Floats.

Por ejemplo, usando una multiplicación en una línea generando 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, enciendo y apaga un LED poco a poco, variando su luminosidad con un bucle **for** :

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

floating Constantes

Similar a constantes enteras, las constantes float son usadas para hacer el código más inteligible.

Ejemplos:

```
n = .005;
```

El tipo Float puede ser expresada en notación científica. 'E' y 'e 'son aceptadas como un indicador de exponente valido.

```
floating-point   equivalente a:      otra equivalencia:
  constantes

   10.0              10
  2.34E5          2.34 * 10^5             234000
  67e-12         67.0 * 10^-12          .000000000067
```

Funciones

Las funciones son segmento de código que permiten al programador crear piezas modulares de código que definen perfectamente una tarea y retornan al área de donde fue llamada esa función. El típico caso de creación de funciones es cuando necesitas realizar una tarea varias veces en un mismo programa.

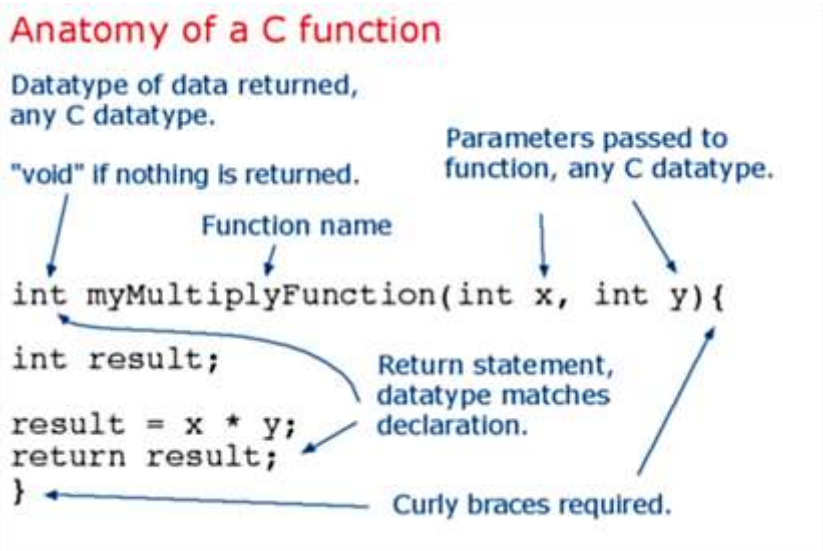
Para programadores acostumbrados a usar BASIC, las funciones proporcionan la funcionalidad de uso de subrutinas (GOSUB en BASIC).

La estandarización del código en las funciones tiene muchas ventajas:

Las funciones ayudan a tener organizado el código. A menudo esto ayuda a entender el programa.
Las funciones codifican una acción en un lugar, así que la función solo es pensada y corregida solo una vez.
Esto reduce los errores en modificaciones del código, si necesita ser cambiado.
Las funciones hacen que el programa sea más compacto, ya que se pueden utilizar en diferentes partes del programa
Esto provoca que el programa sea más modular, y con esto, el programa se hace más ilegible para otros.

Existen dos funciones que son necesarias en Martuino, setup() and loop(). Otras funciones pueden ser creadas fuera de estas dos. Un ejemplo es crear una función simple para multiplicar dos números.

Ejemplo



al "llamar" a nuestra función de multiplicar, pasaremos los parámetros del tipo necesario:

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

```
int j = 3;
int k;

k = myMultiplyFunction(i, j); // k contiene 6
}
```

Nuestra función necesita ser declarada fuera de cualquier otra función, así "myMultiplyFunction()" puede ser definida fuera de la función “loop()" .

El código entero quedará así:

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

void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = myMultiplyFunction(i, j); // k contiene 6
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

Otro ejemplo

Esta función lee un sensor 5 veces con analogRead() y calcula la media de las 5 lecturas. Luego lo escala a valores del rango de 8 bits (0-255), y lo invierte, devolviendo el resultado invertido.

```
int ReadSens_and_Condition(){
  int i;
  int sval = 0;

  for (i = 0; i < 5; i++){
    sval = sval + analogRead(0);    // sensor en pin 0
  }

  sval = sval / 5;    // la media
  sval = sval / 4;    // escala a 8 bits (0 - 255)
  sval = 255 - sval;  // invierte salida
  return sval;
}
```

La llamada de nuestra función la asignamos a una variable.

```
int sens;

sens = ReadSens_and_Condition();
```

goto

Transfiere el puntero del programa a una etiqueta en el programa.

Sintaxis

label:

goto label; // envía el programa a la etiqueta label

Consejos

El uso de goto es rechazado en programación C , y algunos autores de libros de programación en C advierten que el goto nunca es necesario, pero usándolo con mucho cuidado, puede simplificar ciertos programas. La razón por la que los programadores no utilizan esta sentencia es por lo complicado que puede llegar a ser un programa con saltos a diferentes lugares, sin tener un orden y lo difícil que puede resultar entenderlo.

Con esto dicho, usar el comando goto puede venir bien en algunos casos, pero puede llevar a engaño la simplicidad de un programa, sobretodo en bucles.

Ejemplo

```
for(byte r = 0; r < 255; r++){
    for(byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto bailout;}
            // más condiciones ...
        }
    }
}
bailout:
```

highByte()

Descripción

Extrae la parte alta (la más a la izquierda) en tipo byte de un tipo word .

Sintaxis

highByte(x)

Parámetros

x: un valor de cualquier tipo

Retorna

byte

if (condicional) y ==, !=, <, > (comparación)

Veamos el formato de la sentencia if:

```
if (someVariable > 50)
{
  // Haz algo aqui
}
```

El programa testea si la variable someVariable es mayor que 50. Si lo es, el programa hace una determinada acción. Poniéndolo de otra manera. Si lo que se encuentra dentro del paréntesis es verdadero, entrará dentro de las llaves { y ejecutará lo que encuentre dentro. Si no, el programa saltará al siguiente código.

Las llaves {} pueden ser omitidas de una sentencia if. Si esto se hace, la próxima línea de código (definida por el ;) se convierte solo en la parte que se ejecutará si la sentencia es verdadera.

```
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
```

Estas sentencias evaluadas te muestran diferentes formas con diferentes operadores:

Operadores de comparación:

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

Aviso:

Hay que tener cuidado de usar accidentalmente un = simple (ej. `if (x = 10)`). El igual simple "=" es la manera de asignar un valor a una variable. Para comparar una variable con un valor hay que usar doble igual "==" . Si se compara una variable con un solo igual "=" siempre resultará verdadero y siempre se realizará la instrucción de comparación.

Es esto por lo que el lenguaje C evalúa de distinta forma ambas expresiones, si `(x=10)` se evalúa como sigue: 10 es asignado a x (recuerda que el signo es un simple igual que asigna un valor a una variable) así que x ahora contiene 10. Luego el 'if' evalúa 10, que siempre se evalúa a TRUE, Algo que no es cero se evalúa como verdadero TRUE. Consecuentemente, `if (x = 10)` se evaluará como TRUE, y esto no es el valor deseado

cuando usamos un 'if'. Adicionalmente la variable x se le asignará el valor 10, y esto no es deseado ni es lo que pasaría en una comparación, en la cual no se modifica el valor de la variable utilizada.

#include

#include es usado para incluir librarías externas en tu código. Esto da al programador acceso a grupos de librerías estándar de C (gropos de funciones pre-hechas), y además librerías hechas especialmente para Martuino.

Nota: **#include**, es similar a **#define**, en cuanto que no necesitan ";", y si lo pones el compilador dará un mensaje de error.

Ejemplo

Este ejemplo incluye una libraría que se usa para el acceso a la memoria flash del Martuino.

```
#include <SLFS.h>

#include <SPI.h>

#include <Stepper.h>
```

++ (incremento) / -- (decremento)

Descripción

Incrementa o decrementa una variable

Sintaxis

```
x++;  // incrementa x en uno y retorna el viejo valor de x
++x;  // incrementa x en uno y retorna el valor nuevo de x

x-- ;   // decrementa x en uno y retorna el viejo valor de x
--x ;   // decrementa x en uno y retorna el nuevo valor de x
```

Parámetros

x: un entero (int) o long (posiblemente sin signo (unsigned))

Retorna

El original o nuevo valor incrementado / decrementado de la variable.

Ejemplos

```
x = 2;
y = ++x;      // x ahora contiene 3, y contiene 3
y = x--;      // x contiene 2 de nuevo, y contiene 3
```

+= , -= , *= , /=

Descripción

Operación matemática sobre una variable con otra constante o variable. Veamos el significado de cada signo:

Sintaxis

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

Parámetros

x: cualquier tipo de variable

y: cualquier tipo de variable o constante.

Ejemplos

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

int

Descripción

Un entero es el primer tipo de dato almacenado en un valor de 2 bytes. Este está en el rango de -32,768 a 32,767 (el valor mínimo es -2^15 y el máximo valor es (2^15) - 1).

Int almacena números negativos con la técnica llamada complemento a 2. El bit más alto, es conocido como el bit de signo. El complemento a 2 invierte el resto de los bis y le suma 1.

Ejemplo

```
    int ledPin = 13;
```

Sintaxis

```
    int var = val;
```

var – tu nombre de la variable int
val – el valor asignado a la variable

Consejo

Cuando una variable excede su máxima capacidad ellas se dan la vuelta y su valor pasará a ser el mínimo valor posible para esa variable, Evita que esto ocurra en ambas direcciones, tanto por máximo como por el mínimo.

```
   int x
   x = -32,768;
   x = x - 1;       // x ahora contiene 32,767 - se da la vuelta a la
dirección neg.

   x = 32,767;
   x = x + 1;       // x ahora contiene -32,768 - se da la vuelta
```

int()

Descripción

Convierte un valor a tipo int.

Sintaxis

int(x)

Parámetros

x: un valor de cualquier tipo

Retorna

int

Integer Constantes

Las constantes enteras son números que se usan directamente en un programa, como `123`. Por defecto, Estos números son tratados como int's pero puedes cambiarlos con los modificadores U y L.

Normalmente, las constantes enteras son tratadas como enteros en base 10 (decimal) , pero pueden ser usados en otras bases numéricas.

```
Base                Ejemplo    Formato        Comentarios

10 (decimal)            123    ninguno

2 (binario)         B1111011    delante 'B'      solo trabaja con 8 bit
valor (0 to 255)
                                                 caracteres validos 0-1

8 (octal)              0173    delante "0"      caracteres validos 0-7
```

```
16 (hexadecimal)      0x7B    delante "0x"     caracteres validos 0-9,
A-F, a-f
```

Decimal es base 10. Esto es la base de trabajo matemático y con el que estamos más familiarizados. Constantes sin ningún prefijo son asumidas con base decimal.

Ejemplo:

```
101     // mismo que 101 decimal    ((1 * 10^2) + (0 * 10^1) + 1)
```

Binary es base 2. Solo admite caracteres 0 y 1 .

Ejemplo:

```
B101    // mismo que 5 decimal    ((1 * 2^2) + (0 * 2^1) + 1)
```

El formato binario solo trabaja con bytes (8 bits) entre 0 (B0) y 255 (B11111111). Si esto es la entrada de una variable tipo int (16 bits) en forma binaria tu puedes hacerlo en dos pasos, precediendo como sigue:

```
myInt = (B11001100 * 256) + B10101010;    // B11001100 es el byte alto
```

Octal es base 8. Solo admite caracteres desde 0 hasta 7 . Los valores Octal son indicados con un prefijo "0"

Ejemplo:

```
0101    // mismo que 65 decimal    ((1 * 8^2) + (0 * 8^1) + 1)
```

Hay que tener cuidado cuando a una variable le pongas un cero delante, ya que el compilador puede tomarla como un número octal y producir un error no deseado.

Hexadecimal (o hex) es base 16. Son válidos los caracteres desde 0 hasta 9 y las letras A hasta F; A equivale a un valor 10, B es 11, y así hasta la F, con valor 15. Los valores Hex son indicados con el prefijo "0x". Nota: las letras A-F pueden utilizarle con mayúsculas o minúsculas (a-f).

Ejemplo:

```
0x101   // mismo que 257 decimal    ((1 * 16^2) + (0 * 16^1) + 1)
```

U y L formatos

Por defecto, una constante de número entero es tratada como un int con sus limitaciones en sus valores. Las especificaciones de las constantes de números enteros o cualquier otro tipo son como sigue:

una 'u' o 'U' fuerza a la constante a tener un formato sin signo. Ejemplo: `33u`
una 'l' o 'L' fuerza a la constante a tener un formato de long. Ejemplo: `100000L`
una 'ul' o 'UL' fuerza a la constante a tener un formato long sin signo. Ejemplo: `32767ul`

interrupts()

Descripción

Re-habilita las interrupciones (que puedan ser deshabilitadas con noInterrupts()). Las interrupciones permiten ciertas tareas importantes que ocurren en segundo plano y están habilitadas por defecto. Algunas funciones no trabajarán mientras las interrupciones están deshabilitadas, y las comunicaciones pueden ser ignoradas. Las interrupciones pueden modificar ligeramente el tiempo de duración del código, ahora bien, es posible deshabilitarlas para secciones particulares de nuestro código.

Parámetros

Ninguno

Retorna

Nada

Ejemplo

```
void setup() {}

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

long

Descripción

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

Ejemplo

```
    long speedOfLight = 186000L;   // ver las constantes de números
enteros para entender la letra 'L'
```

Sintaxis

```
    long var = val;
```

var El nombre de la variable Long
val – el valor asignado a la variable

long()

Descripción
Convierte un valor a tipo long .

Sintaxis
long(x)

Parámetros
x: un valor de cualquier tipo

Retorna
long

loop()
Después de crear la función setup() , para inicializar y poner valores iniciales a las variables , la función loop() hace lo que precisamente dice su nombre , un bucle, permitiendo que tu programa cambie y responda. Úsalo para el control de la placa Martuino.

Ejemplo

```
int buttonPin = PUSH2;

// inicializa pueto serie y el pin con el botón
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT_PULLUP);
}

// bucle que testea si el botón cambia de estado cada vez,
// y envía por el puerto serie si está pulsado
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    Serial.write('H');
  else
```

```
    Serial.write('L');

  delay(1000);
}
```

lowByte()

Descripción

Extrae el byte bajo (el de más a la derecha) de una variable (ej. De un word).

Sintaxis

lowByte(x)

Parámetros

x: un valor de cualquier tipo

Retorna

byte

Llaves {}

Las llaves se utilizan en la mayor parte del lenguaje de programación en C . Son utilizadas en diferentes construcciones y esto a veces puede confundir a los principiantes en este lenguaje.

Una llave abierta "{" siempre será seguida de una llave cerrada "}". Esto es una condición Esta es una condición que es a menudo referida como llaves equilibradas. El IDE (integrated development environment) de Martuino incluye una conveniente característica para chequear el equilibrio de llaves.

Los programadores novatos, y los programadores que llegan a C desde BASIC a menudo se encuentran confundidos o desalentados. Después de todo, las llaves reemplazan la declaración RETURN en una subrutina (función), la declaración ENDIF en una condición y la declaración NEXT en un bucle FOR .

Porque el uso de las llaves son tan variadas, es buena práctica de programación que se escriba la llave de cerrar “}” inmediatamente después de escribir la llave abierta “{“ . Luego algunos insertan un retorno de carro (un intro) entre la llave de abrir y la de cerrar.

El principal uso de las llaves {}

Funciones

```
void myfunction(argumento){
  statements(s)
}
```

bucles

```
while (boolean expresion)
{
   statement(s)
}

do
{
   statement(s)
} while (boolean expression);

for (inicialización; terminación condición; incrementar expr)
{
   statement(s)
}
```

Declaración Condicional

```
if (boolean expresion)
{
   statement(s)
}

else if (boolean expresion)
{
   statement(s)
}
else
{
   statement(s)
}
```

map(valor, fromLow, fromHigh, toLow, toHigh)

Descripción

Re-mapea un número desde un rango a otro. Esto es, un **valor** de **fromLow** será mapeado a **toLow**, un valor de **fromHigh** a **toHigh**, de valores entre a valores entre , etc.

No se limitan los valores dentro del rango, porque los valores fuera de rango son a veces pretendidos y útiles. En la función constrain() es posible usarlos ya sea antes o después de esta función, si los límites de los rangos son deseados.

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

La siguiente función maneja bien los números negativos:

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

es válido y funciona bien.

La función map() usa números enteros, así que no genera fracciones, las fracciones son truncadas y el resultado no es redondeado ni promediado.

Parámetros

valor: El número a mapear.

fromLow: El rango mínimo del valor actual

fromHigh: el rango máximo del valor actual

toLow: el rango mínimo del valor de destino

toHigh: el rango máximo del valor de destino

Retorna

El valor mapeado.

Ejemplo

```
/* Mapea el valor analógico 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 matemáticos aplicados, aquí hay una 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;
}
```

max(x, y)

Descripción

Calcula el máximo de dos números.

Parámetros

x: el primer número de cualquier tipo

y: el segundo número de cualquier tipo

Retorna

El mayor de los dos valores

Ejemplo

```
sensVal = max(senVal, 20); // asigna sensVal al valor máximo de
sensVal o 20
                           // (así me aseguro que el máximo valor es
20)
```

Nota

Quizás intuitivamente, max() se utiliza a menudo para restringir el extremo inferior de un rango de una variable, mientras que min() se utiliza a menudo para restringir el extremo superior del rango.

Aviso

Durante el uso de la función max() , evita utilizar otras funciones dentro de los paréntesis, puede dar resultados incorrectos:

```
max(a--, 0);   // evita esto - da resultados incorrectos

a--;           // usa esto -
max(a, 0);     // mantiene intacta la función
```

micros()

Descripción

Retorna el número de microsegundos desde que la placa Martuino se puso a funcionar con el actual programa. Este número se sobrepasa y vuelve a cero después de aproximadamente 70 minutos.

Nota: 1.000 microsegundos es un milisegundo y 1.000.000 de microsegundos es un segundo.

Parámetros
Ninguno

Retorna
Número de microsegundos desde que el programa se inició (*unsigned long*)

Ejemplo
```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = micros();
  //muestra el tiempo desde el principio
  Serial.println(time);
  // espera un segundo antes de volver a enviar la información
  delay(1000);
}
```

millis()

Descripción
Retorna el número de milisegundos desde que la placa Martuino se puso a funcionar con el actual programa. Este número se sobrepasa y vuelve a cero después de aproximadamente 50 días.

Parámetros
Ninguno

Retorna
Número de milisegundos desde que el programa comenzó (*unsigned long*)

Ejemplo
```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();
  //muestra el tiempo desde que empezó
  Serial.println(time);
  // espera un segundo antes de volver a enviar la info
  delay(1000);
}
```

Consejo:

El parámetro es un unsigned long, se producirá un error si el programador intenta hacer operaciones matemáticas con tipos como números enteros, ints.

min(x, y)

Descripción

Calcula el mínimo de dos números

Parámetros

x: el primer número de cualquier tipo

y: el segundo número de cualquier tipo

Retorna

El más pequeño de los dos números

Ejemplo

```
sensVal = min(sensVal, 100); // asigna sensVal al menor de sensVal o
100
                             // te aseguras que sensval no tomará
valores superiores de 100.
```

Nota

Quizás intuitivamente, max() se utiliza a menudo para restringir el extremo inferior de un rango de una variable, mientras que min() se utiliza a menudo para restringir el extremo superior del rango.

Aviso

Durante el uso de la función min() , evita utilizar otras funciones dentro de los paréntesis, puede dar resultados incorrectos:

```
min(a++, 100);   // evitar esto – resultados incorrectos

a++;
min(a, 100);    // usa esto
```

% (módulo)

Descripción

Calcula el resto de una división entre números enteros. Esto se utiliza para mantener una variable dentro de un rango particular (ej. El tamaño de un array).

Sintaxis

resultado = dividendo % divisor

Parámetros

dividendo: el número a dividir

divisor: el número que divide

Retorna

El resto de la división

Ejemplos

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

Ejemplo con código

```
/* actualiza el valor de un array cada vez que se produce el bucle*/

int values[10];
int i = 0;

void setup() {}

void loop()
{
  values[i] = analogRead(A0);
  i = (i + 1) % 10;   // el resto opera sobre la variable
}
```

Consejo

El módulo no trabaja con variables de tipo floats.

noInterrupts()

Descripción

Deshabilita interrupciones (puedes re-habilitarlas con interrupts()). Las Interrupciones permiten ciertas tareas importantes que ocurren en segundo. Algunas

funciones del sistema no funcionarán si deshabilitas las interrupciones como pueden ser las comunicaciones, ignorando los datos que entran. Las interrupciones pueden alargar el tiempo del programa y podrían deshabilitarse en partes del programa que fuera critico el tiempo de realización de ese código.

Parámetros

Ninguno.

Retorna

Nada.

Ejemplo

```
void setup() {}

void loop()
{
  noInterrupts();
  // critica, código sensible al tiempo de duración
  interrupts();
  // otro código aqui
}
```

noTone()

Descripción

Para la generación de una onda cuadrada (tono) puesta en marcha por el comando tone(). No hace nada si no hay ningún tono generado.

NOTA: Si quieres producir diferentes tonos en múltiples pines, necesitas llamar a noTone() en cada pin antes de llamar a tone() en el próximo pin.

Sintaxis

noTone(pin)

Parámetros

pin: el pin donde se parará el tono

Retorna

Nada

operador +

Descripción

Combina, o concatena dos variables de tipo string. El segundo tipo string es añadido al primero, y el resultado se sitúa en otra variable tipo string. Trabaja igual que string.concat().

Sintaxis

string3 = string1 + string 2; string3 += string2;

Parámetros

string, string2, string3: variables de tipo String

Retorna

Una nueva variable de tipo String con la combinación de los dos originales strings.

Operador = asignar (signo igual simple)

Almacena el valor del lado derecho del signo = en la variable a la izquierda del signo =.

En programación en lenguaje C el signo sencillo = se utiliza para asignar un valor. Es diferente que el signo igual en algebra que significa igualdad. Este operador lo utiliza el microcontrolador para asignar un valor en el lado derecho del signo = a una variable a lado izquierdo del signo igual.

Ejemplo

```
 int sensVal;                 // declara un entero llamada sensVal
 senVal = analogRead(0);       // almacena el valor (digitalizado) de
la entrada analógica del pin 0
```

Consejos de programación

La variable a la izquierda del operador (=) necesita estar habilitada para poder almacenar un valor en ella. Declara el tipo de variable a utilizar antes de asignarle un valor.

No confundas el operador [=] (signo igual simple) con el de comparación [==] (doble signo igual), no se evaluarán las dos expresiones por igual.

operador ==

Descripción

Compara si son iguales dos variables de tipo String. Ten en cuenta que el string "hola" no es igual a "HOLA". Funcionalmente es lo mismo que string.equals()

Sintaxis

string1 == string2

Parámetros

string1, string2: variables de tipo String

Retorna

true: si ambas string son iguales
false: si no lo son.

Operador puntero

& (referencia) and * (dereferencia)

Los punteros son unas de la cosas más difíciles para los principiantes en C, y es posible que no encuentres punteros en la mayoría de código para Martuino. Pero la manipulación de ciertas estructuras de datos, usan punteros para simplificar el código, y esto hace necesario saber que los punteros son una buena herramienta.

Operador Boolean

Esto puede usarse dentro de una instrucción if .

&& (and lógica)

True si ambos operadores son true

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

es true solo si ambas están a nivel alto(HIGH).

|| (or lógico)

True si cualquiera de ellas es true.

```
if (x > 0 || y > 0) {
  // ...
}
```
es true si x o y son mayores que 0.

! (not)
True si el operador en false.

```
if (!x) {
  // ...
}
```
es true si x es false (ej. si x = 0).

Aviso
Augúrate de no equivocarte con el operador boleano AND , o doble ampersand && con el operador AND o simple ampersand & . Son enteramente diferentes.

Igualmente no confundas el operador boleano || (doble pipe) con el operador OR | (simple pipe).

El not ~ (tilde) es muy diferente al operador boleano not ! (exclamación o "bang" como dicen los programadores). Debes de asegurarte de utilizar el que desees en cada momento.

Ejemplo
```
if (a >= 10 && a <= 20){}   // true si esta entre 10 y 20
```

Operaciones Aritméticas:

Suma, Resta, Multiplicación, y División

Descripción
Estos operadores retornan la suma, diferencia, producto, o división respectivamente de dos operandos. La operación es realizada usando el tipo de dato del operando, así por ejemplo, `9 / 4 da 2` si 9 y 4 son de tipo ints (números enteros). Esto hay que tenerlo en cuenta, porque un producto de dos números puede ser mayor que el tipo de dato asignado al resultado, produciendo errores, el resultado no será el esperado . Por ejemplo, si sumaos 1 a un int con un valor de 32,767 da -32,768). Si los operadores son de diferente tipo el más grande será usado para almacenar el resultado.

Si uno delos operandos es de tipo **float** o de tipo **double**, el tipo float será usado para el resultado de la operación.

Ejemplos

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

Sintaxis

```
resultado = valor1 + valor2;
resultado = valor1 - valor2;
resultado = valor1 * valor2;
resultado = valor1 / valor2;
```

Parámetros

valor1: una variable o constante

valor2: una variable o constante

Consejos:

Sabiendo que los integer constants por defecto son int, algunos cálculos puede sobrepasar el límite de este tipo de dato (ej. 60 * 1000 dará un resultado negativo). Elije variables de tamaño mayor para almacenar los resultados de las operaciones Debes de conocer en qué punto tu variable se dará la vuelta, es decir, pasará de un número positivo a número negativo o viceversa, por ejemplo, 0 – 1 0 0 - - 32768. Para operaciones con fracciones utiliza el tipo float, pero ten en cuenta los inconvenientes de operaciones de gran tamaño, la velocidad de computación disminuye.

pinMode()

Descripción

Configura el especificado pin como entrada o salida. Échale un vistazo al funcionamiento de los pines digitales.

Sintaxis

pinMode(pin, mode)

Parámetros

pin: el número de pin que utilizaremos.

mode: será INPUT(entrada) o OUTPUT(salida) o INPUT_PULLUP(entrada con resistencia interna a +vcc)

Retorna

Nada

Ejemplo

```
int ledPin = 14;                  // LED conectado a pin digital 14

void setup()
{
  pinMode(ledPin, OUTPUT);       // pon el pin 14 como salida
}

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

Nota

Las pines analógicos pueden usarse como pines digitales, referenciados como A0,A1,A2 o 16,17 y 18.

pow(base, exponente)

Descripción

Calcula el valor de un número elevado a una potencia. Pow() puede usarse para elevar un número a una fracción de potencia. Esto se utiliza para generar valores exponenciales o trazado de curvas.

Parámetros

base: el número (*float*)

exponente: la potencia a la que será elevado (*float*)

Retorna

El resultado de la potencia (*double*)

pulseIn()

Descripción

Lee un pulso (HIGH o LOW) de un pin. Por ejemplo, si el **valor** es **HIGH**, **pulseIn()** esperará a que el pin sea **HIGH**, pasará un tiempo, y luego esperará a que el pin sea **LOW** y parará el tiempo. Retornará la longitud del pulso en microsegundos. El retorno será cero si el pulso comienza sin un especifico tiempo máximo para producirse.

El tiempo de esta función ha sido puesto empíricamente y es probable que de errores para pulsos muy largos. Trabaja con pulsos entre 10 microsegundos y 3 minutos de longitud.

Sintaxis

pulseIn(pin, valor)
pulseIn(pin, valor, timeout)

Parámetros

pin: el número de pin donde quieres leer el pulso. (*int*)

valor: tipo de pulso a leer: HIGH o LOW. (*int*)

timeout (opcional): el número de microsegundos de espera para que comience el pulso, por defecto es un segundo(*unsigned long*)

Retorna

La longitud del pulso (en microsegundos) o 0 si el pulso no ha comenzado después del tiempo de espera (timeout) de tipo *unsigned long*.

Ejemplo

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

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

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

random()

Descripción

Este función genera un pseudo-número aleatorio.

Sintaxis

random(max)
random(min, max)

Parámetros

min – el menor del valor del número aleatorio, incluido este número *(opcional)*

max – el mayor del valor del número aleatorio, excluido del valor.

Retorna

Un número aleatorio entre un mínimo y un máximo-1 (*long*)

Nota:

Si es importante que una secuencia de números generada por random() sea diferente, utiliza la instrucción randomSeed() para inicializar el generador de números aleatorios con bastante aleatoriedad, como un analogRead() de un pin no conectado a nada.

Si tú necesitas repetir una secuencia aleatoria exactamente puedes llamar a randomeed() con un número fijo, antes de empezar la secuencia aleatoria.

Ejemplo

```
long randNumber;

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

  // si la entrada analógica pin 0 esta desconectada, el ruido
  // analógico causará que la llamada a randomSeed() genere
  // diferentes números cada vez que ejecutes este código.
  // randomSeed() será quien baraje la función random().
  randomSeed(analogRead(0));
}

void loop() {
  // muestra números aleatorios desde 0 a 299
  randNumber = random(300);
  Serial.println(randNumber);

  // muestra números aleatorios desde 10 a 19
  randNumber = random(10, 20);
  Serial.println(randNumber);

  delay(50);
}
```

randomSeed(seed)

Descripción

randomSeed() inicializa el generador aleatorio de pseudo-números, causando que comience la secuencia de números arbitrariamente. Esta secuencia, mientras sea muy larga, y aleatoria, será siempre igual.

Si es importante que una secuencia de números generada por random() sea diferente, utiliza la instrucción randomSeed() para inicializar el generador de números aleatorios con bastante aleatoriedad, como un analogRead() de un pin no conectado a nada.

Si tú necesitas repetir una secuencia aleatoria exactamente puedes llamar a randomeed() con un número fijo, antes de empezar la secuencia aleatoria.

Parámetros

Seed – número que genera la semilla de la secuencia aleatoria (long,int).

Retorna

Nada

Ejemplo

```
long randNumber;

void setup(){
  Serial.begin(9600);
  randomSeed(1);
}

void loop(){
  randNumber = random(300);
  Serial.println(randNumber);

  delay(50);
}
```

return

Termina una función y retorna un valor a la función desde donde fue llamada, si se desea.

Sintaxis:

return;

return valor; // ambas son validas

Parámetros

valor: cualquier variable o constante.

Ejemplo:

Una función que compara una entrada analógica con un to a límite:

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

Acuérdate de utilizar comentarios para saber, en este caso, que el número de retorno indica algo en concreto.

```
void loop(){

// Código para testear

return;

// el resto del código aqui
// nunca será ejecutado, por estar detrás del return;
}
```

; coma

Usado como final de una instrucción.

Ejemplo

```
int a = 13;
```

Consejo

Si olvidas un punto y coma ";" al final de una línea el compilador dará un error. El texto de error será obvio, y hará referencia a la pérdida de un punto y coma, o quizás no. El compilador testea primero si está el punto y coma, pero a veces puede dar el error en líneas de código anteriores o que están después del error.

Serie comunicaciones

Esta guía te ayudará con las comunicaciones con el puerto serie de Martuino

El puerto serie se le suele llamar UART **(**Universal Asynchronous Receiver-Transmitter) de sus siglas en ingles

Funciones de puerto serie

available()
begin()
end()
find()
findUntil()
flush()
parseFloat()
parseInt()
peek()
print()
println()
read()
readBytes()
readBytesUntil()
setTimeout()
write()

available()

Descripción

Coge el número de bytes (caracteres) disponibles para leer en el puerto serie. Esta dato se habrá recibido y almacenado en el buffer receptor (que posee 128 bytes). available() se hereda de la clase Stream .

Sintaxis

Serial.available()

Parámetros

Ninguno

Retorna

Número de bytes disponible para leer.

Ejemplo

```
int incomingByte = 0;   // para los bytes del puerto serie

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

void loop() {

        // envía algún dato solo cuando recibas algo:
        if (Serial.available() > 0) {
                // lee los datos recibidos:
                incomingByte = Serial.read();

                // muestra que los has recibido:
                Serial.print("I received: ");
                Serial.println(incomingByte, DEC);
        }
}
```

begin()

Descripción

Configura la velocidad del puerto serie (baudios) para transmisión d datos. Para comunicaciones con computador, usa una de estas velocidades: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. Puedes, si lo deseas, especificar otras velocidades - por ejemplo, comunicaciones sobre pines 0 y 1 con componentes que requieran velocidades particulares.

Sintaxis

Serial.begin(speed)

Parámetros

speed: en bits por segundo (baud) - *long*

Retorna

Nada

Ejemplo:

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

end()

Descripción

Deshabilita comunicaciones, permitiendo que los pines utilizados RX y TX puedan ser usados para propósitos general, entradas o salidas. Para rehabilitar las comunicaciones utiliza Serial.begin().

Sintaxis
Serial.end()

Parámetros
Ninguno

Retorna
Nada

find()

Descripción
Serial.find() lee datos del buffer serie y busca un carácter en la longitud total del buffer. La función retorna true si encontró el/los caracteres y, false si no. find() es heredada de la clase Stream

Sintaxis
Serial.find(*target*)

Parámetros
target : carácter a buscar (char)

Retorna
boolean

findUntil()

Descripción
Serial.findUntil() lee datos del buffer serie y busca un carácter o la terminación por toda la longitud del buffer. Esta función retorna true si el carácter es encontrado y, false si no. Serial.findUntil() es heredada de la clase Stream .

Sintaxis
Serial.findUntil(*target, terminal*)

Parámetros
target : carácter a buscar (char)

terminal : carácter de terminación a buscar (char)

Retorna
boolean

flush()

Descripción
Limpia el buffer de recepción de los datos serie. Cualquier llamada a Serial.read() o Serial.available() retornan los datos recibidos después de la última llamada de Serial.flush().

Serial.flush() espera la salida de los datos almacenados en el buffer antes de limpiar el buffer.

flush() es heredada de la clase Stream .

Sintaxis
Serial.flush()

parseFloat()

Descripción
Serial.parseFloat() retorna el primer carácter de tipo float del buffer de recepción. Los caracteres que no son dígitos (o el signo menos) son saltados. parseFloat() es terminada por el primer carácter que no sea del tipo float. Serial.parseFloat() es heredada de la clase Stream .

Sintaxis
Serial.parseFloat()

Parámetros
Ninguno

Retorna
float

parseInt()

Descripción
Mira el próximo dato valido del tipo entero en el buffer de recepción. parseInt() es heredada de la clase Stream .

Sintaxis
Serial.parseInt()

Parámetros
Ninguno

Retorna
int : el próximo dato de tipo entero.

peek()

Descripción

Retorna el próximo byte (carácter) del buffer de recepción sin remover lo que hay dentro del buffer serie. Esto es, con la llamada a peek() devolverá el mismo carácter, que con la próxima llamada a read(). peek() es heredada de la clase Stream .

Sintaxis

Serial.peek()

Parámetros

Ninguno

Retorna

El primer byte del buffer receptor del puerto serie (o -1 si no hay datos disponibles) - *int*

print()

Descripción

Muestra los datos del puerto serie en texto utilizando el código ASCII. Este comando puede tomar muchas formas. Los números son mostrados usando la tabla ASCII de 8 dígitos. Los datos de tipo float son mostrados similarmente con la tabla ASCII, por defecto con dos decimales. Bytes son enviados como carácter simple. Caracteres y datos de tipo string son enviados como son. Por ejemplo:

Serial.print(78) muestra "78"
Serial.print(1.23456) muestra "1.23"
Serial.print('N') muestra "N"
Serial.print("Hello world.") muestra "Hello world."

Se puede especificar un segundo parámetro, la base (formato) a usar; permite valores en BIN (binario, o base 2), OCT (octal, o base 8), DEC (decimal, o base 10), HEX (hexadecimal, o base 16). Para datos de tipo float, este parámetro especifica el número de decimales a mostrar. Por ejemplo:

Serial.print(78, BIN) muestra "1001110"
Serial.print(78, OCT) nuestra "116"
Serial.print(78, DEC) muestra "78"
Serial.print(78, HEX) muestra "4E"
Serial.println(1.23456, 0) muestra "1"
Serial.println(1.23456, 2) muestra "1.23"
Serial.println(1.23456, 4) muestra "1.2346"

Tu puedes pasar caracteres desde la memoria flash a Serial.print() envolviéndolo con F(). Por ejemplo :

Serial.print(F(â€œHello Worldâ€□))

Para enviar un byte simple, usa Serial.write().

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

Parámetros
val: valor a mostrar, de cualquier tipo

format: Especifica la base numérica (enteros, string) o número de decimales a mostrar(float)

Retorna
byte
print() retorna el número de bytes escritos, aunque leyendo el número es opcional.

Ejemplo:

```
/*
  Usa un bucle FOR para mostrar los números de varios formatos.
*/
int x = 0;    // variable

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

void loop() {
  // print labels
  Serial.print("NO FORMAT");       // muestra la etiqueta
  Serial.print("\t");              // muestra una tabulación

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

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

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

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

  for(x=0; x< 64; x++){

    // muéstralo en cualquier formato:
    Serial.print(x);       // muestra en decimal ASCII igual que con
"DEC"
    Serial.print("\t");    // muestra una tabulación

    Serial.print(x, DEC);  // muestra en decimal ASCII
    Serial.print("\t");    // muestra una tabulación

    Serial.print(x, HEX);  // muestra en hexadecimal ASCII
    Serial.print("\t");    // muestra una tabulación
```

```
    Serial.print(x, OCT);  // muéstralo en Octal ASCII
    Serial.print("\t");    // muestra una tabulación

    Serial.println(x, BIN);  // muéstralo en binario ASCII
    //                            luego añade el retorno de carro con
"println"
    delay(200);            // espera 200 milisegundos
  }
  Serial.println("");      // muestra un retorno de carro
}
```

Consejos

Todas las transmisiones serie son asíncronas; Serial.print()retornará antes de que cualquier carácter sea trasmitido.

println()

Descripción

Muestra los datos del puerto serie en texto utilizando el código ASCII seguido de un carácter de retorno de carro (ASCII 13, o '\r') y un carácter de salto de línea(ASCII 10, o '\n'). Este comando tomas las mismas formas que Serial.print().

Sintaxis

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

Parámetros

val: El valor a mostrar, de cualquier tipo

format: Especifica la base numérica (enteros, string) o número de decimales a mostrar(float)

Retorna

byte
println()retorna el número de bytes escritos, aunque leyendo el número es opcional

Ejemplo:

```
/*
  Entrada analógica

  Lee la entrada analógica, y muestra el valor.

  Creado el 24 Marzo de 2006
  de Tom Igoe
*/

int analogValue = 0;    // variable para el valor
void setup() {
  // abre el puerto serie a 9600 bps:
  Serial.begin(9600);
```

```
}

void loop() {
  // lee la entrada analógica:
  analogValue = analogRead(0);

  // muéstrala en varios formatos:
  Serial.println(analogValue);       // decimal
  Serial.println(analogValue, DEC);  // decimal
  Serial.println(analogValue, HEX);  // hexadecimal
  Serial.println(analogValue, OCT);  // octal
  Serial.println(analogValue, BIN);  // binario

  // retardo de 10 milisegundos antes de otra lectura:
  delay(10);
}
```

read()

Descripción

Lee la entrada de datos del puerto serie. data. read() es heredada de la clase Stream .

Sintaxis

Serial.read()

Parámetros

Ninguno

Retorna

El primer dato recibido disponible (o -1 sino hay datos disponibles) - *int*

Ejemplo

```
int incomingByte = 0;

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

void loop() {

        // envía un dato solo cuando reciba otro:
        if (Serial.available() > 0) {
                // lee los datos:
                incomingByte = Serial.read();

                // muestra lo recibido en decimal:
                Serial.print("I received: ");
                Serial.println(incomingByte, DEC);
        }
}
```

readBytes()

Descripción

Serial.readBytes() lee caracteres del buffer receptor del puerto serie. Esta función termina cuando se encuentra el carácter de fin de buffer, o se han leído datos de una determinada longitud, o ha pasado un tiempo demasiado grande (ver Serial.setTimeout()). Serial.readBytes() retorna el número de caracteres situados en el buffer. Un 0 indica que no hay datos válidos. Serial.readBytes() es heredada de la clase Stream .

Sintaxis

Serial.readBytes(buffer, length)

Parámetros

buffer: el buffer donde se almacenan los datos (de tipo char[] o byte[])

length : el número de bytes a leer (int)

Retorna

byte

readBytesUntil()

Descripción

Serial.readBytesUntil() lee los caracteres del buffer receptor serie y los mete dentro de una array. Esta función termina si el carácter de fin de buffer es encontrado, o determinada longitud leída, o paso un determinado tiempo determinado por el comando Serial.setTimeout()). Serial.readBytesUntil() retorna el número de caracteres que hay en el buffer. -1 indica que no encontró datos válidos. Serial.readBytesUntil() es heredada de la clase Stream .

Sintaxis

Serial.readBytes(character, buffer, length)

Parámetros

character : el carácter a buscar (char)

buffer: el buffer donde almacenar los datos (de tipo char[] o byte[])

length : el número de bytes a leer (int)

Retorna

byte

setTimeOut()

Descripción
Determina el tiempo máximo que se esperará la recepción de caracteres a través del puerto serie. Serial.setTimeOut()es heredada de la clase Stream .

Sintaxis
Serial.setTimeout(time)

Parámetros
time : duración en milisegundos (long).

Retorna
Nada

write()

Descripción
Escribe datos binarios en el puerto serie. Estos datos son enviados como byte o una serie de bytes; para enviar caracteres que representen dígitos numéricos utiliza la función print().

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

Parámetros
val: un valor que se envía como un byte

str: un tipo string que se envía como una serie de bytes

buf: un array que se envía como serie de bytes

len: la longitud del buffer

Retorna
byte
write() retornará el número de bytes escritos, aunque la lectura del número es opcional

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

void loop(){
  Serial.write(45); // envía un byte con valor 45
```

```
   int bytesSent = Serial.write("hello"Â□); //envía un string
"hello"Â□ y retorna la longitud del string.
}
```

**********Hasta aquí puerto serie**********

setup()

La función setup() es utilizada cuando comienza el programa. Utilízala para inicializar variables, los modos de los pines, uso de librerías, etc. La función setup se ejecuta solo una vez, cada vez que arranques el Martuino o la resetees.

Ejemplo

```
int buttonPin = PUSH2;

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

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

shiftIn()

Descripción

Lee de un pin los datos, desplazándolos una a uno y dejándolos en un byte de datos. Comienza por el bit más significativo(el más a la izquierda) o el menos (el más a la derecha). Por cada bit, la línea de reloj es puesta a nivel alto, el próximo bit es leído de la línea de datos, y después es puesta a nivel bajo la línea de reloj.

Nota: Este software esta implementado en una librería, llamada SPI que usa el hardware, mucho más rápido pero solo trabaja con pines específicos.

Sintaxis

shiftIn(dataPin, clockPin, bitOrder)

Parámetros

dataPin: el pin por donde se leerá cada bit (*int*)

clockPin: el pin que hará las funciones de reloj cuando se envíen los datos por el **dataPin** (*int*)

bitOrder: orden de llegada de los bits; **MSBFIRST** o **LSBFIRST**.
(Más Significativo Bit primero, o, Menos Significativo Bit primero)

Retorna

El valor leído por el DataPin (*byte*)

shiftOut()

Descripción

Escribe un byte en pin desplazándolos uno a uno. Comienza con el bit más significativo (el más a la izquierda) o el menos significativo (el más a la derecha). Para cada bit generará un pulso en el pin de reloj (sube a nivel alto y luego lo baja) para indicar que el dato está disponible.

Nota: Si está conectando con un aparato que necesita los datos en el pulso alto del reloj, asegúrate que la línea de datos está a nivel bajo antes de llamar a shiftOut(), por ejemplo con el comando digitalWrite(clockPin, LOW).

Este software esta implementado en una librería, llamada SPI , que usa el hardware mucho más rápido pero con pines específicos.

Sintaxis

shiftOut(dataPin, clockPin, bitOrder, valor)

Parámetros

dataPinel pin por donde saldrán los datos (*int*)

clockPin: el pin que hará las funciones de reloj cuando se envíen los datos por el **dataPin** (*int*)

bitOrderorden de salida delos bits; **MSBFIRST** o **LSBFIRST**.
(Más Significativo Bit primero, o, Menos Significativo Bit primero)

valor : el dato a enviar. (*byte*)

Retorna

Nada

Nota

El **dataPin** y **clockPin** han de estar configurados como salida con el comando pinMode().

shiftOut permite sacar un byte (8 bits) si así lo requieres, un valor en decimal de hasta 255. Si necesitas valores superiores lo puedes hacer en dos pasos como el ejemplo:

```
// haz primer MSBFIRST
int data = 500;
// saca el byte alto
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// saca el byte bajo
shiftOut(data, clock, MSBFIRST, data);

// o LSBFIRST
data = 500;
// saca el byte bajo
shiftOut(dataPin, clock, LSBFIRST, data);
// saca el byte alto
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

sin(rad)

Descripción

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

Parámetros

rad: el ángulo en radianes (*float*)

Retorna

El seno del ángulo (*double*)

Símbolo AND (&), símbolo OR (|), símbolo XOR (^)

Símbolo AND (&)

Este operador funciona a nivel de bit dentro de una variable. Estos te resolverán un rango de problemas de programación.

Descripción y Sintaxis

A continuación detallaremos la descripción y sintaxis de estos operadores.

Símbolo AND (&)

El símbolo AND en C++ es una simple ampersand, &, usada entre dos expresiones de números enteros. Operador AND trabaja a nivel de cada bit independientemente, de acuerdo con esta norma: Si ambos bits son 1,el resultado es 1, al contrario la salida es 0. Otra forma de expresar esto es:

```
    0   0   1   1    operand1
    0   1   0   1    operand2
    ----------
```

```
    0   0   0   1     (operand1 & operand2) - resultado devuelto
```

En Martuino, el tipo de entero (int) es de 16-bit , así usando & entre dos enteros (int) da como resultado 16 simultaneas ANDs .En este fragmento de código se ve mejor:

```
    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 a y en b son procesados usando el operador AND, y el resultado de los 16 bits son almacenados en c, resultando el valor 01000100 en binario, 68 en decimal.

Uno de los casos en que más comúnmente se usa el operador AND es para seleccionar un particular bit (o bits) desde un valor entero, o también llamada máscara. Ver el ejemplo anterior.

Símbolo OR (|)

El símbolo OR en C++ es una barra vertical, |. Como el operador &, | opera independientemente en cada bit en ambas expresiones enteras, pero este operador lo hace diferente. El operador OR trabaja de acuerdo a esto: si ambos bits o tan solo uno de los bits está a 1 la salida será 1, al contrario será 0. En otras palabras:

```
    0   0   1   1     operand1
    0   1   0   1     operand2
    ----------
    0   1   1   1     (operand1 | operand2) - resultado
```

Este es un ejemplo del operador OR usado en el código C++ :

```
    int a =  92;    // en binario: 0000000001011100
    int b = 101;    // en binario: 0000000001100101
    int c = a | b;  // resultado:  0000000001111101, o 125 en decimal.
```

Programa ejemplo

Un trabajo común de los operadores AND y OR es que el programador lea-modifique-escriba en un puerto de entradas o salidas. En los microcontroladores, un puerto es de 8 bits representando el valor de cada pin de ese puerto. Al escribir en un puerto se modifican todos los pines de ese puerto.

P1OUT es una constante construida para hacer referencia al estado de salida de los pines digitales 0,1,2,3,4,5,6,7. Si el pin de una posición en concreto está a 1, el pin estará a HIGH (nivel alto). (Este pin necesita haberse puesto como salida con el comando pinMode().) Así si escribimos `P1OUT = B00110001;` hemos hecho que los pines 2,3 y 7 este a HIGH. En este caso nosotros no queremos cambiar el estado de los pines 1 y 2,

que corresponden a las comunicaciones serie, y podríamos interferir en dichas comunicaciones. Para evitar esta interferencia necesitamos preservar el estado de los bits 1 y 2 con la siguiente máscara: B00000110.

```
     Nuestro algoritmo de programa será:
```

Toma P1OUT y limpia solo los bits correspondientes a los pines que queremos controlar (con el operador AND).
Con la modificación del valor de P1OUT aplicamos un nuevo valor a los pines que vamos a controlar(con el operador OR).
Pin 1 y Pin 2 serán reservados para las comunicaciones serie.

```
int i;     // contador
int j;

void setup(){
P1DIR = P1DIR | B11111001; // pon a "1" los pines 2 al 7,
//como están como salida pinMode(pin, OUTPUT) los pines 2 al 7
Serial.begin(9600);
}

void loop(){
for (i=0; i<64; i++){

P1OUT = P1OUT & B00000110;  // limpia los bits 2 - 7, sin tocar los
demás (xx & 11 == xx)
j = i;
bit = j & 1; // Conserva bit 0
j = (j << 2);   // desplaza los pines 3 - 7 y 0 -  evitando que los
pines 1 y 2 varíen.
j = j | bit;  // OR entre los desplazados 1 - 7 con bit 0
P1OUT = P1OUT | j;          // Combina el puerto con la nueva
información de los LED
Serial.println(P1OUT, BIN); // depura y muestra es estado de P1OUT
delay(100);
   }
}
```

símbolo XOR (^)

Este símbolo no es usado habitualmente en C++ , EXCLUSIVE OR, conocido como símbolo XOR. El operador XOR se representa con el símbolo ^. Este operador es muy similar al símbolo OR |, solo tendrá una salida a 0 cuando ambos bits de entrada sean 1:

```
    0  0  1  1    operand1
    0  1  0  1    operand2
    ----------
    0  1  1  0    (operand1 ^ operand2) - resultado
```

Otra manera de ver el símbolo XOR es que cada bit tiene un resultado a 1 si cada entrada correspondiente tiene los bits diferentes, o a 0 que es lo mismo.

Un simple ejemplo:

```
    int x = 12;     // binario: 1100
```

```
    int y = 10;      // binario: 1010
    int z = x ^ y;   // binario: 0110, o decimal 6
```

El operador ^ es usado a menudo para cambiar el estado de un bit de una expresión numérica entera (por ejemplo, cambia deo a 1, o 1 a 0) . En una operación XOR si la máscara es un 1 ,ese bit es invertidos es un 0, el bit no es invertido y permanece igual. Veamos cómo hacer parpadear un pin:

```
// Parpadeo_Pin_0
// demo para Exclusive OR
void setup(){
P1DIR = P1DIR | B000000001; // pin 0 como salida
}

void loop(){
P1OUT = P1OUT ^ B00000001;  // invierte bit 0 (digital pin 0), los
demás no se tocan
delay(100);
}
```

Símbolo compuesto AND (&=)

Descripción

El símbolo compuesto del operador AND (&=) es utilizado para forzar un particular bit a estado bajo LOW (a 0) de una variable o una constante.

Sintaxis:

```
x &= y;   // equivalente a x = x & y;
```

Parámetros

x: una variable de tipo char, int o long
y: una constante de tipo char, int, o long

Ejemplo:

Primero, échale un vistazo al operador AND (&)

```
   0  0  1  1    operando1
   0  1  0  1    operando2
   ----------
   0  0  0  1    (operando1 & operando2) - resultado
```

Los Bits que con "operador ANDed" a 0 son limpiados a 0 , Si myByte es una variable de tipo byte,

```
myByte & B00000000 = 0;
```

Los Bits con "operador ANDed" a 1 no serán cambiados,

```
myByte & B11111111 = myByte;
```

Nota: Para trabajar a nivel de bits con estos operadores – es conveniente usar el formato binario con constantes. El número es el mismo pero en otra representación, pero es más fácil entenderlo. B00000000 es mostrado para aclarar mejor el formato binario, pero verdaderamente es un cero "0".

Consecuentemente – para limpiar bits (poner a Zero) 0 y 1 en una variable, y no modificar el resto de la variable, usa el operador compuesto AND (&=) con la constante B11111100

```
   1  0  1  0  1  0  1  0    variable
   1  1  1  1  1  1  0  0    mascara
   ----------------------
   1  0  1  0  1  0  0  0

 variable no cambiada
                    bits limpios
```

Ahora veremos lo mismo con los bits reemplazados por el símbolo x

```
   x  x  x  x  x  x  x  x    variable
   1  1  1  1  1  1  0  0    mascara
   ----------------------
   x  x  x  x  x  x  0  0

 variable no cambiada
                    bits limpios
```

Así si:

```
myByte =  10101010;

myByte &= B1111100 == B10101000;
```

Símbolo compuesto OR (|=)

Descripción

El operador compuesto OR (|=) se usa cuando queremos poner bits particulares a "1" de una variable.

Sintaxis:

```
x |= y;   // equivalente a x = x | y;
```

Parámetros

x: una variable de tipo char, int o long
y: una contante de tipo char, int, o long

Ejemplo:

Primero échale un vistazo al operador OR (|)

```
   0  0  1  1    operando1
   0  1  0  1    operando2
   ----------
   0  1  1  1    (operando1 | operando2) - resultado
```

Los Bits que "bitwise ORed" con un 0 no cambiarán, así si la variablemyByte, myByte | B00000000 = myByte;

Los Bits que "bitwise ORed" con 1 serán puesto a 1:
myByte | B11111111 = B11111111;

Consecuentemente – para poner los bits 0 y 1 de una variable, mientras el resto de la variable no cambia, utiliza el operador compuesto OR (|=) con la constante B00000011

```
   1  0  1  0  1  0  1  0    variable
   0  0  0  0  0  0  1  1    mascara
   ----------------------
   1  0  1  0  1  0  1  1

 variable sin cambios
                    bits puestos a 1
```

Ahora mostraremos lo mismo reemplazando los bits de la variable por el símbolo x

```
   x  x  x  x  x  x  x  x    variable
   0  0  0  0  0  0  1  1    mascara
   ----------------------
   x  x  x  x  x  x  1  1

 variable sin cambios
                    bits puestos a 1
```

Así si:

```
myByte =  B10101010;

myByte |= B00000011 == B10101011;
```

Símbolo NOT (~)

El operador NOT en el lenguaje C++ es el carácter ~. A diferencia de & y |, El operador NOT es aplicado como un simple operador. El operador NOT cambia cada bit al bit opuesto, o lo hace 1 y 1 lo hace 0. Por ejemplo:

```
 0  1    operando1
----------
 1  0   ~ operando1
 int a = 103;    // binario:  0000000001100111
 int b = ~a;     // binario:  1111111110011000 = -104
```

Podría sorprenderte que el resultado de esta operación sea un número negativo como -104 . Esto ocurre porque el bit más alto de una variable tipo entero(int) es llamado bit de signo. Si el bit más alto (el de la izquierda) está a 1, el número es interpretado como un número negativo. Esta codificación de números positivos y negativos es referenciada en el complemento a 2, que ya hemos explicado con anterioridad

Puede resultar interesante saber que en una variable de tipo entero x, ~x es lo mismo que -x-1.

A veces, el bit de signo en números enteros puede causar alguna no deseada sorpresa.

sizeof

Descripción

Este operador devuelve el número de bytes en un variable, o el número de bytes que ocupa un array.

Sintaxis

sizeof(variable)

Parámetros

variable: cualquier variable o array (ejemplo int, float, byte)

Ejemplo

Este operador es usado sobre todo con arrays (como strings) cuando es conveniente cambiar el tamaño de un array sin perjudicar a otras partes del programa.

Este ejemplo muestra un carácter cada vez y su valor de un string. Prueba a cambiar el texto del string.

```
char myStr[] = "this is a test";
int i;
```

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

void loop() {
  for (i = 0; i < sizeof(myStr) - 1; i++){
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.println(myStr[i], BYTE);
  }
}
```

Nota: el sizeof devuelve el total de números de bytes. Así si la variable es muy grande para que sea del tipo int, el bucle loop debería ser algo así:

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)) - 1; i++) {
  // algo con myInts[i]
}
```

sqrt(x)

Descripción

Calcula la raíz cuadrada de un número.

Parámetros

x: el número, de cualquier tipo de dato.

Retorna

double, el resultado de la raíz cuadrada.

Static

La palabra static es usada para crear variables que son visibles solo dentro de una función. Sin embargo a diferencia de las variables locales que son creadas y destruidas cada vez que la función donde se encuentran es llamada, las variables static persisten más allá de las llamada a la función, persistiendo su dato entre llamadas a la función.

Variables declaradas como static Solo serán creadas e inicializadas la primera vez que es llamada la función donde se encuentran.

Ejemplo

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk deambula arriba y abajo aleatoriamente entre dos
* puntos. El máximo movimiento en un bucle (loop) es governado por
* el parametro "stepsize".
* una variable static es movida arriba y abajo una cantidad aleatoria.
* Esta técnica es conocida como "pink noise" y "drunken walk".
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;
int total;

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

void loop()
{        //  testea randomWalk función
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
   delay(10);
}

int randomWalk(int moveSize){
  static int  place;     // variable para almacenar random walk -
declarada static
                         // valor entre llamadas a esta función , pero
no otras funciones pueden variar este valor
  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange){                    // testea el
límite por debajo y arriba
    place = place + (randomWalkLowRange - place);     // refleja el
número en dirección positiva
  }
  else if(place > randomWalkHighRange){
    place = place - (place - randomWalkHighRange);     // refleja el
número en dirección negativa
  }
```

```
  return place;
}
```

Stream

Stream es una clase para caracteres y datos binarios. No es llamada directamente, pero es invocada cuando usas una función que se basa en esto.

Stream define las funciones de lectura en Martuino. Cuando usas cualquier funcionalidad de lectura o algún método similar, puedes sin peligro asumir llamadas a la clase Stream. Para funciones como print(), Stream hereda desde la clase Print.

Entre las librerías que confían en Stream se incluyen:

Serial
Wire

Funciones

available()
read()
flush()
find()
findUntil()
peek()
readBytes()
readBytesUntil()
parseInt()
parsefloat()
setTimeout()

available()

Descripción

available() toma el número de bytes disponibles en un stream. Esto es solo para bytes que hayan llegado al stream.

Esta función es parte de la clase Stream , y es llamada por cualquier clase de estas (Wire, Serial, etc).

Sintaxis

stream.available()

Parámetros

stream : una estancia de una clase heredada de Stream.

Retorna

int : número de bytes disponibles para leer.

find()

Descripción

find() lee los datos de un stream hasta que encuentra un string como el buscado. La función devuelve verdadero si encuentra el string o false si llega al final.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.find(target)
target : el string a buscar como (char)

Parámetros

stream : una estancia de una clase heredad de Stream.

Retorna

boolean

findUntil()

Descripción

findUntil() lee datos de un stream hasta que encuentra un string de una dada longitud o un string de terminación es encontrado. Esta función retorna verdadero si el string es encontrado o falso si llega al final.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.findUntil(target,terminal)
target : el string a buscar como (char)

Parámetros

stream : una estancia de la clase heredada de Stream.
target el string a buscar como (char)
terminal : el string terminal donde buscar como (char)

Retorna

boolean

flush()

Descripción

flush() limpia un buffer cuando todos los caracteres han sido enviados.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.flush()

Parámetros

stream : una estancia de una clase heredada de Stream.

Retorna

boolean

parseFloat()

Descripción

parseFloat() retorna el primer número de tipo float de la posición actual. Inicialmente los caracteres que no son dígitos (o de signo menos) son saltados. parseFloat() termina por el primer carácter que no es un número de tipo float.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.parseFloat(list)

Parámetros

stream : una estancia de una clase heredada de Stream.
list : el stream testeada con floats (char)

Retorna

float

parseInt()

Descripción

parseInt() retorna el primer número entero valido tipo (long)desde la posición actual. Inicialmente los caracteres que no son números enteros (o de signo menos) son saltados. parseInt termina por el primer carácter que no es un digito.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.parseInt(list)

Parámetros

stream : una estancia de una clase heredada de Stream.
list : el stream a testear con ints (char)

Retorna

int

peek()

Descripción

Lee un byte desde un archivo sin avanzar al próximo. Esto es, que sucesivas llamadas a peek() devolverán el mismo valor, siendo la próxima llamada a Read().

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.peek()

Parámetros

stream : una estancia de la clase heredada de Stream.

Retorna

El próximo byte (o carácter), o -1 si no está disponible.

read()

Descripción

read() lee un carácter entrante a un stream de un buffer.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.read()

Parámetros

stream : una estancia de una clase heredada de Stream.

Retorna

El primer byte entrante disponible (o -1 si no hay datos disponibles)

readBytes()

Descripción

readBytes() lee caracteres desde un stream dentro de un buffer. La función termina si una determina longitud ha sido leída, o si finalizo el tiempo máximo. readBytes() devuelve el número de caracteres localizados en un buffer.

Esta función es parte de la clase Stream, y es llamada por las clases heredadas como (Wire, Serial, etc).

Sintaxis

stream.readBytes(buffer, length)

Parámetros

stream una estancia de una clase heredada de Stream.
buffer: el buffer que almacenará los bytes de tipo (char[] o byte[])
length : el número de bytes a leer (int)

Retorna

byte

ReadBytesUntil()

Descripción

readBytesUntil() lee los caracteres de un stream dentro de un buffer. La función terminará si detecta el carácter de fin, o si llega al final de la longitud del buffer, o en caso que el tiempo máximo de espera se haya sobrepasado(ver setTimeout()). readBytesUntil() retorna el número de caracteres en el buffer. Un 0 ocurre cuando no hay datos válidos.

Este función es parte de la clase Stream,y es llamada por cualquier clase heredad de (wire,Serial, etc). Ver la claseStream para más información.

Sintaxis

stream.readBytesUntil(character, buffer, length)

Parámetros

stream : una instancia heredada de la clase Stream.
character : el carácter a buscar (char)
buffer: el buffer donde se almacena los datos (char[] o byte[])
length : el número de bytes a leer (int)

Retorna

byte

setTimeout()

Descripción

setTimeout() es el máximo tiempo de espera para un Stream, por defecto es de 1000 milisegundos.

Este función es parte de la clase Stream,y es llamada por cualquier clase heredad de (wire,Serial, etc). Ver la claseStream para más información.

Sintaxis

stream.setTimeout(time)

Parámetros

stream : una instancia de la clase heredada de Stream.
time : duración en milisegundos (long).

Retorna

Nada

string

Descripción

Los textos pueden ser representados de dos maneras. Puedes usar el tipo de dato String , o puedes hacer un texto desde un array que contenga datos de tipo char seguido de un carácter de fin.

Ejemplos

Todas las siguientes declaraciones son buenas para String.

```
char Str1[15];
char Str2[8] = {'e', 'n', 'e', 'r', 'g', 'i', 'a'};
char Str3[8] = {'e', 'n', 'e', 'r', 'g', 'i', 'a', '\0'};
char Str4[ ] = "energia";
char Str5[8] = "energia";
char Str6[15] = "energia";
```

Posibilidades para declarar strings

Declarando un array de datos tipo chars sin inicializarlo como en Str1
Declarando un array de datos tipo chars (con un char extra) y el compilador añadirá el carácter de fin (null carácter), como en Str2
Explícitamente añadir el carácter de fin (null carácter), Str3
Inicializarlo con una constante entre comillas; el compilador le dará el tamaño correcto al array y le añadirá el carácter de fin (null carácter) al final, Str4
Inicializar el array con un explícito tamaño y constante, Str5
Inicializar el array, añadiendo espacio extra para un texto más grande, Str6

Carácter de fin (Null termination)

Generalmente, un tipo string es terminado con el carácter de fin(en ASCII un cero “0”). Esto permite a las funciones (como Serial.print()) saber dónde finaliza el string. Por el contrario, si no existe el carácter de fin continuaría leyendo bytes de memoria que no formarían parte del string.

De esta manera un string necesita espacio para contener un carácter más que el texto que desees que contenga. Esto es porque Str2 y Str5 necesitan 8 caracteres, aunque contengan "energia" de solo 7 – la última posición es automáticamente ocupada por un carácter de fin (null carácter). Str4 tendrá automáticamente el tamaño de 8 caracteres, uno para el extra null. En Str3, incluimos el carácter de fin (null termination) implícitamente escribiendo '\0' nosotros mismos.

Es posible tener un string sin carácter de fin (null carácter) (si tú has especificado la longitud de Str2 no como 7 si no como 8) al darle el tamaño del texto que contendrá + 1.

Simples comillas o dobles comillas?

Los Strings siempre estarán definidos por dobles comillas ("Abc") y los caracteres siempre estarán definidos por simples comillas ('A').

Envasando largos strings

Tu puedes envasar largos string como sigue:

```
char myString[] = "Esta es mi primera línea"
" Esta es la segunda"
" etcétera";
```

Arrays de strings

A menudo esto es un inconveniente, Cuando se trabaja con largas cadenas de texto, como cuando se trabaja con un display LCD, prepararemos un array. Porque un string por sí mismo es un array, esto es un ejemplo de un array bi-dimensional.

En el código más abajo, el asterisco después de tipo de datos char "char*" indica que es un puntero de un array . Todos los nombres de array son punteros, así que estos son necesarios para hacer un array de arrays. Los punteros es una de las cosas más exotéricas del lenguaje C para su comprensión por principiantes.

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);
   }
}
```

String()

Descripción

Constructor de una instancia de la clase String. Hay múltiples versiones de constructores para Strings desde diferentes tipos de datos, incluyendo:

Una constante de múltiples caracteres, en doble comillas (ej. Un array char)
Una constante de un simple carácter, una simple comilla
Otra instancia de un objeto String
Una constante de número enteros o tipo long
Una constante de número enteros o tipo long, usando una específica base numérica
Una variable con un número entero o tipo long
Una variable con un número entero o tipo long, usando una específica base numérica

Construyendo un String desde un número resulta que es un string que contiene la representación de ese número en la tabla ASCII. Por defecto se utiliza la base 10

```
String thisString = String(13)
```

El String contiene un "13". Puedes usar otras bases numéricas. Por ejemplo,

```
String thisString = String(13, HEX)
```

El string contiene una "D", como representación hexadecimal, en decimal contendrá un valor 13. O si lo prefieres en binario,

```
String thisString = String(13, BIN)
```

El string contendrá "1101", como representación en binario que es el número 13.

Sintaxis

String(val)
String(val, base)

Parámetros

val: una variable con formato String - *string*, *char*, *byte*, *int*, *long*, *unsigned int*, *unsigned long*
base (*opcional*) – la base numérica de la variable.

Retorna

Un a instancia de la clase String

Ejemplos

Todas las declaraciones siguientes son válidas:

```
String stringOne = "Hello String";                       // usando una
constante String
```

String

Descripción

La clase String , permite el uso y manipulación de texto en su manera más compleja como carácter arrays. Puedes concatenar Strings, añadir a ellos, buscar para reemplazar una parte, y más. Este ocupa más memoria que un simple array de caracteres, pero es más usado.

Por referencia, los arrays de caracteres son referenciados con un strings con una minúscula, e instancias de la clase String son referenciadas con la S mayúscula. Las constantes string son especificadas con dobles comillas y no son instancias de la clase String.

Funciones

String()
charAt()
compareTo()
concat()
endsWith()
equals()
equalsIgnoreCase()
getBytes()
indexOf()
lastIndexOf()
length()
replace()
setCharAt()
startsWith()
substring()
toCharArray()
toLowerCase()
toUpperCase()
trim()

Operadores

[] (element access)

String.charAt()

Descripción

Acceso a un particular carácter de un String.

Sintaxis

string.charAt(n)

Parámetros

string: una variable de tipo string

n: el carácter a acceder

Retorna

La posición donde se encuentra ese carácter dentro del String

String.compareTo()

Descripción

Compara dos Strings, testeando si uno viene antes o después de otro, o si son iguales. Los strings son comparados carácter a carácter, usando los valores ASCII de los caracteres. De manera que, por ejemplo, la 'a 'va antes que la 'pero después de la 'A'. Los números van antes de las letras (Mirar la tabla ASCII que esta con anterioridad).

Sintaxis

string.compareTo(string2)

Parámetros

string: una variable de tipo string

string2: otra variable de tipo String

Retorna

Un número negativo: si string va antes que string2
0: si string es igual a string2
un número positivo: si string va después de string2

String.concat()

Descripción

Combina, o *concatena dos* strings sobre un nuevo String. El segundo string es añadido al primero, y el resultado se coloca en el nuevo String.

Sintaxis

string.concat(string, string2)

Parámetros

string, string2: variables de tipo string

Retorna
Nuevo string que combina los dos strings, string y string2

String.endsWith()

Descripción
Testea sí o no un String finaliza con un carácter de otro String.

Sintaxis
string.endsWith(string2)

Parámetros
String: una variable de tipo string

string2: otro variable de tipo string

Retorna

true: si el string finaliza con los caracteres de string2
false: lo contrario

String.equals()

Descripción
Compara dos string para ver si son iguales. La comparación se hace respetando mayúsculas y minúsculas, de manera que “hola” no es igual que “HOLA”.

Sintaxis
string.equals(string2)

Parámetros
string, string2: variables de tipo String

Retorna

true: si string es igual a string2
false: lo contrario

String.equalsIgnoreCase()

Descripción

Compara dos strings para ver si son iguales. La comparación no diferencia entre mayúsculas y minúsculas, de manera que “hola” si es igual a “HOLA”.

Sintaxis

string.equalsIgnoreCase(string2)

Parámetros

string, string2: variables de tipo String

Retorna

true: si string es igual a string2 (ignorando mayúsculas y minúsculas)
false: lo contrario

String.getBytes()

Descripción

Copia los caracteres de un string a un buffer.

Sintaxis

string.getBytes(buf, len)

Parámetros

string: una variable de tipo string

buf: el buffer donde se copiarán los caracteres(*byte []*)

len: el tamaño del buffer (*unsigned int*)

Retorna

Nada

String.indexOf()

Descripción

Localiza un carácter o String dentro de otro String. Por defecto, buscará desde el principio del String, pero puede dársele un índice, permitiendo la localización de todas las instancias de caracteres o String.

Sintaxis

string.indexOf(val)
string.indexOf(val, from)

Parámetros

string: una variable de tipo string

val: el valor a buscar de tipo - *char* o *String*
from: el índice a empezar con la búsqueda

Retorna

El índice dentro del string donde localizo el carácter o string o -1 si no lo encontro.

String.lastIndexOf()

Descripción

Localiza un character o String dentro de otro String. Por defecto, la búsqueda se hace desde el final del String, pero puede trabajar desde un índice dado, permitiendo localizar todas las instancias de caracteres o String.

Sintaxis

string.lastIndexOf(val)
string.lastIndexOf(val, from)

Parámetros

string: una variable de tipo string
val: el valor a buscar de tipo - *char* o *String*
from: el índice a finalizar con la búsqueda

Retorna

El índice dentro del string donde localizo el carácter o string o -1 si no lo encontró.

String.length()

Descripción

Retorna la longitud de un string, en caracteres. (Nota la longitud no incluye el carácter de fin (null carácter).)

Sintaxis

string.length()

Parámetros

string: una variable de tipo string

Retorna

La longitud de caracteres de un string

String.replace()

Descripción

La función replace() de string permite reemplazar toda una instancia de unos caracteres dados por otros. Puedes usar replace para reemplazar un aparte de un string (substring) por otro diferente substring.

Sintaxis

string.replace(substring1, substring2)

Parámetros

string: una variable de tipo string
substring1: otra variable de tipo String
substring2: otra variable de tipo String

Retorna

otro String que contiene el nuevo string con caracteres reemplazados.

String.setCharAt()

Descripción

Pon un carácter en un String. No tiene efecto si indicas un índice más grande que la longitud del string.

Sintaxis

string.setCharAt(index, c)

Parámetros

string: una variable de tipo string
index: el índice donde colocar el carácter
c: el carácter a almacenar en la dada localización

Retorna

Nada

String.startsWith()

Descripción

Testea sí o no empieza un String con un carácter u otro String.

Sintaxis

string.startsWith(string2)

Parámetros

string, string2: variables de tipo String

Retorna

true: si string empieza por los caracteres de string2
false: lo contrario

String.substring()

Descripción

Toma un substring de un string. El principio del índice incluye el correspondiente carácter del substring, pero el final del índice no incluye el carácter del substring. Si el final del índice es omitido, el substring continua hasta el final del String.

Sintaxis

string.substring(from)
string.substring(from, to)

Parámetros

string: una variable de tipo string
from: el índice donde empieza el substring
to (*opcional*): el índice donde finaliza el substring -1

Retorna

el substring

String.toCharArray()

Descripción

Copia el carácter de un string al proporcionado buffer.

Sintaxis

string.toCharArray(buf, len)

Parámetros

string: una variable de tipo string

buf: el buffer donde se copiará el carácter (*char []*)

len: el tamaño del buffer (*unsigned int*)

Retorna

Nada

String.toLowerCase()

Descripción

Toma la versión en minúsculas de un String. toLowerCase() modifica el string en lugar de retornar otro nuevo.

Sintaxis

string.toLowerCase()

Parámetros
string: una variable de tipo string

Retorna
Nada

String.toUpperCase()

Descripción
Toma la versión mayúsculas de un string. toUpperCase() modifica el string en lugar de retornar uno nuevo.

Sintaxis
string.toUpperCase()

Parámetros
string: una variable de tipo string

Retorna
Nada

trim()

Descripción
Toma la versión de un string quitando los espacios que encuentre dentro del string. trim() modifica el string en lugar de retornar uno nuevo.

Sintaxis
string.trim()

Parámetros
string: una variable de tipo string

Retorna
Nada

[] (acceso a un elemento)

Descripción

Permite acceder a un individual carácter de un string.

Sintaxis

char thisChar = string1[n]

Parámetros

char thisChar – una variable de tipo char
string1 – una variable de tipo string
int n – un a variable de tipo numérica

Retorna

El carácter correspondiente al número n dentro de un string. Igual que charAt().

switch / case

Como la instrucción **if** , **switch...case** controla el flujo de un programa permitiendo a los programadores diferentes códigos que serán ejecutados según varias condiciones. En particular, una instrucción switch compara el valor de una variable a los valores especificados en las instrucciones case. Cuando en una instrucción case es encontrada una coincidencia con la variable que hay en switch, el código que hay en la instrucción case se ejecuta.

El comando **break** hará que se salga de la instrucción switch , y es usado típicamente para finalizar cada instrucción case. Sin un comando break, la instrucción switch continuará ejecutándose hasta la siguiente expresión hasta encontrar un break, o el fin de la instrucción switch.

Ejemplo

```
switch (var) {
  case 1:
    //algo que hacer cuando vale  1
    break;
  case 2:
    //algo que hacer cuando vale 2
    break;
  default:
    // Si no hay ninguna coincidencia, haz lo por defecto
    // por defecto es opcional, no es necesario ponerlo
}
```

Sintaxis

```
switch (var) {
  case label:
    // código
    break;
  case label:
    // código
    break;
  default:
    //código
}
```

Parámetros

var: la variable que se comparará en los varios cases

label: un valor a comparar con la variable de switch

tan(rad)

Descripción

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

Parámetros

rad: el ángulo en radianes (*float*)

Retorna

La tangente del ángulo (*double*)

tone()

Descripción

Genera una onda cuadrada de una específica frecuencia(y un 50% del ciclo de trabajo) en un pin. La duración puede ser especificada, si no la onda continuará hasta que llamemos a noTone(). El pin puede ser conectado a un buzzer piezo-electrico o algún otro altavoz para ser escuchado el tono.

Solo un tono puede ser generado a la vez. Si un tono está ejecutándose en un diferente pin, al llamar a tone() no tendrá efecto. Si el tono está ejecutándose en el mismo pin, la llamada a tone() cambiará la frecuencia al nuevo tono.

NOTA: si quieres ejecutar diferentes tonos en múltiples pines, necesitas llamar a noTone() en cada pin antes de llamar a tone() en el próximo pin.

Sintaxis

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

Parámetros

pin: el pin donde se generará el tono

frequency: la frecuencia del tono en hercios - *unsigned int*

duration: la duración del tono en milisegundos(opcional) - *unsigned long*

Retorna

Nada

unsigned char

Descripción

Un tipo de dato sin signo que ocupa 1 byte de memoria. Lo mismo que un tipo de dato byte .

El tipo de dato char sin signo contiene números desde 0 a 255.

Entre los programadores se prefiere el tipo de dato byte.

Ejemplo

```
    unsigned char myChar = 240;
```

unsigned int

Descripción

Números enteros sin signo son los mismo que ints que ocupan 2 bytes. Al contrario que estos, no admiten números negativos, solo positivos, por lo tanto el rango e uso de este tipo será desde 0 a 65,535 (2^16) - 1).

La diferencia de los ints sin signo y los ints con signo (normales, int), es el bit más alto, el llamado bit de signo. En Martuino el tipo int (con signo), si el bit más alto es un "1", el número se interpreta como un número negativo y los otros 15 bits están en complemento a 2.

Ejemplo

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

Sintaxis

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

var – el nombre de la variable sin signo
val – el valor asignado a esa variable

Consejos

Si en el valor de una variable se sobrepasa su máxima capacidad, llamado "roll over" o su mínima capacidad, puede ocurrir en ambas direcciones, podemos obtener valores no deseados. Intenta que esto no ocurra y tenlo en cuenta.

```
   unsigned int x
   x = 0;
   x = x - 1;        // x ahora contiene 65535 - rolls over en neg
dirección
   x = x + 1;        // x ahora contiene 0 - rolls over
```

unsigned long

Descripción

Las variables long sin signo son variables extendidas en su tamaño para almacenar números de hasta 32 bits (4 bytes). A diferencia de los longs los longs sin signo no almacenan números negativos, haciendo su rango desde 0 a 4,294,967,295 (2^32 - 1).

Ejemplo

```
unsigned long time;

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

void loop()
{
  Serial.print("Time: ");
  time = millis();
  //muestra el tiempo desde que empezó el programa
  Serial.println(time);
  // espera un segundo para no enviar muchos datos continuos
  delay(1000);
}
```

Sintaxis

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

var – el nombre de tu variable long
val – el valor asignado a esa variable

Variables

Una variable es una manera de llamar y almacenar un valor para más tarde usarlo en el programa, como el dato de un sensor o un valor intermedio de un cálculo.

Declarando Variables

Antes de que sean usadas, todas las variables tienen que ser declaradas. Declararlas es una manera de definir un tipo de variable, y opcionalmente, ponerles un valor inicial (inicialización de una variable). Las Variables no tienen por qué ser inicializadas cuando son declaradas (asignarles un valor) , pero se suele hacer.

```
int inputVariable1;
int inputVariable2 = 0;     // ambas son correctas
```

Los Programadores consideran el tamaño de los números para elegir el tipo de variables. Las Variables sobrepasarán su valor máximo cuando excedan del espacio asignado para su almacén en memoria.

Alcance de una Variable

Otra elección importante del programador es el lugar donde declarar variables. El especifico lugar en el que las variables serán declaradas influye en como varias funciones en un programa verán esa variable. Esto es llamado el alcance de una variable.

Inicializando Variables

Las Variables pueden ser inicializadas (asignarles u valor al principio) cuando ellas sean declaradas o no. Es una práctica de un buen programador validar el dato que contendrá la variable, antes que sea utilizada para algún otro propósito.

Ejemplo:

```
 int calibrationVal = 17;  // declara calibrationVal y dale un valor
inicial
```

Sobrepasar el valor de una Variable

Cuando las variables son sobrepasadas por un valor mayor que el de su capacidad o por el mínimo de su capacidad, el valor de las variables puede no ser el esperado.

```
   int x
   x = -32,768;
   x = x - 1;       // x ahora contiene 32,767 - rolls over en
dirección neg.
```

```
   x = 32,767;
   x = x + 1;       // x ahora contiene -32,768 - rolls over
```

Usando las Variables

Una variables tiene que ser declarada, luego le pondremos un valor con el signo = que será almacenado dentro de esa variable. El nombre de la variable estará situado al lado izquierdo del signo igual y el valor asignado estará al lado derecho del signo =.

```
inputVariable1 = 7;              // pon un nombre a una variable
inputVariable1 to 7
inputVariable2 = analogRead(2); // pon el nombre inputVariable2
                                 // (digitaliza) la tensión leída desde
la entrada analógicapin #2
```

Ejemplos

```
 int lightSensVal;
   char currentLetter;
   unsigned long speedOfLight = 186000UL;
   char errorMessage = {"choose another option"}; // ver string
```

Cuando a una variable se le ha dado un valor(asignado un valor),puedes testear ese valor para ver si cumple ciertas condiciones, o puedes usar su valor directamente. El siguiente código testea si la variable inputVariable2 es mayor de 100, luego el retardo que usará esa variable tendrá un mínimo de 100:

```
if (inputVariable2 < 100)
{
  inputVariable2 = 100;
}

delay(inputVariable2);
```

Este ejemplo muestra el uso de operaciones con variables. Se testea la variable (`if (inputVariable2 < 100)`), si la variable es menor de 100 se le asigna el valor de 100 (`inputVariable2 = 100`), y se usa el valor de esa variable como parámetro de un retardo con la función delay() (`delay(inputVariable2)`)

Nota: dale a tus variables nombres descriptivos de lo que van a hacer o a contener. Variables con nombre como **Sensorhumedad** o **pushButton** te ayudan (y a cualquiera que lea el código) a entenderlo. Las Variables llamadas **var** o **valor**, no ayudan a entender el código.

Puedes nombrar a las variables con cualquier palabra que no sea una de los comandos descritos en este manual. Evita empezar nombres de variables con caracteres numéricos.

Ajunos tipos de variable

char
byte
int
unsigned int
long
unsigned long
float
double

volatile

volatile es un comando conocido como variable *índice*, es usada para antes de que una variable tenga un tipo de dato, modificarla de manera que el compilador y las secuencias del programa traten la variable.

Declarando una variable volatile es una directiva para el compilador. El compilador es el software que traduce tu código C/C++ al lenguaje máquina, que es realmente el que entiende Martuino.

Específicamente, dirige al compilador a cargar la variable desde la RAM y no desde el registro de almacenamiento, mientras esta en una localización de memoria temporal donde el programa almacena y manipula las variables. Bajo ciertas condiciones, el valor de una variable es almacenada en un registro que puede ser incorrecto.

Una variable será declarada volatile cuando este valor pueda ser cambiado por algo más allá del control de la sección de código que aparece, como cuando ejecutamos un hilo (thread). En Martuino, esto solo puede ocurrir en la sección de código asociada con interrupciones, llamada rutina de servicio de interrupciones.

Ejemplo

```
volatile int state = HIGH;
volatile int flag = HIGH;
int count = 0;

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

  pinMode(GREEN_LED, OUTPUT);
  digitalWrite(GREEN_LED, state);
  attachInterrupt(PUSH2, blink, FALLING); // la interrupción es
ejecutada cuando es presionado el botón
}

void loop()
{
  digitalWrite(GREEN_LED, state); //LED encendido
```

```
  if(flag) {
    count++;
    Serial.println(count);
    flag = LOW;
  }
}

void blink()
{
  state = !state;
  flag = HIGH;
}
```

while bucle

Descripción

Bucles while son bucles continuos, e infinitos, hasta el la expresión que hay dentro de los paréntesis, () se hace falsa (false). A veces no cambiará la variable que se testea, y el bucle **while** nunca se acabará. Esto podría pasar en tu código, como un incremento de una variable, o una condición externa, como testear un sensor.

Sintaxis

```
while(expression){
  // código a ejecutar
}
```

Parámetros

expression - un (boolean) C que se evaluará a true o false

Ejemplo

```
var = 0;
while(var < 200){
  // algo que se repite 200 veces
  var++;
}
```

word

Descripción

Una variable de tipo Word almacena un número de 16-bit sin signo, desde 0 a 65535. Lo mismo que un tipo int sin signo.

Ejemplo

```
word w = 10000;
```

word()

Descripción

Convierte un valor a tipo word o crea un dato de dos bytes.

Sintaxis

word(x)
word(h, l)

Parámetros

x: un valor de cualquier tipo

h: El byte alto (el de la izquierda) de la variable word

l: el byte bajo (el de la derecha) de la variable word

Retorna

word