UNIT 4

Syllabus

Introduction to Arduino programming, Structures sketch, control structure, further syntax, Arithmetic operators, Comparison operators, Boolean operators, pointer access operators, bitwise operators, compound operators. Variables constants, data types, variable scope & qualifiers, utilities, conversion, Functions digital I/O, analog I/O, advanced I/O, time, math, trigonometry, characters, random numbers, bits & bytes, external interrupts, interrupts, communication, USB.

Introduction to Arduino programming:

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing.

Arduino programming language can be divided in three main parts:

1. Functions,
2. Values (variables and constants), and
3. Structures.

Structures:

The elements of Arduino (C++) code.

Sketch, Control Structure, Further Syntax, Arithmetic Operators, Comparison Operators, Boolean Operators, Pointer Access Operators, Bitwise Operators, Compound Operators.

Sketch

setup():

Description: The setup() function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc. The setup() function will only run once, after each powerup or reset of the Arduino board.

Example Code

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

loop() :

Description: After creating a setup() function, which initializes and sets the initial values, the loop() function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.

Example Code

```
int buttonPin = 3;
// setup initializes serial and the button pin
void setup() {
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}
// loop checks the button pin each time,
// and will send serial if it is pressed
void loop() {
  if (digitalRead(buttonPin) == HIGH) {
    Serial.write('H');
  }
  else {
    Serial.write('L');
  }
  delay(1000);
}
```

Control Structure

If:

Description: The if statement checks for a condition and executes the proceeding statement or set of statements if the condition is 'true'.

Syntax

```
if (condition) {
  //statement(s)
}
```

Parameters

condition: a boolean expression (i.e., can be true or false).

Example Code

The brackets may be omitted after an if statement. If this is done, the next line (defined by the semicolon) becomes the only conditional statement.

```
if (x > 120)
digitalWrite(LEDpin, HIGH);
```

else:

Description: The if…else allows greater control over the flow of code than the basic if statement, by allowing multiple tests to be grouped. An else clause (if at all exists) will be executed if the condition in the if statement results in false. The else can proceed another if test, so that multiple, mutually exclusive tests can be run at the same time.

Each test will proceed to the next one until a true test is encountered. When a true test is found, its associated block of code is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default else block is executed, if one is present, and sets the default behaviour.

Note that an else if block may be used with or without a terminating else block and vice versa. An unlimited number of such else if branches are allowed.

Syntax

```
if (condition1) {
  // do Thing A
}
else if (condition2) {
  // do Thing B
}
else {
  // do Thing C
}
```

Example Code

Below is an extract from a code for temperature sensor system

```
if (temperature >= 70) {
  // Danger! Shut down the system.
}
else if (temperature >= 60) { // 60 <= temperature < 70
  // Warning! User attention required.
}
else { // temperature < 60
  // Safe! Continue usual tasks.
}
```

for:

Description: The for statement is used to repeat a block of statements enclosed in curly braces. An increment counter is usually used to increment and terminate the loop. The for statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

Syntax

```
for (initialization; condition; increment/decrement) {
  // statement(s);
}
```

Parameters

initialization: happens first and exactly once.
condition: each time through the loop, condition is tested; if it's true, the statement block, and the **increment** is executed, then the **condition** is tested again. When the **condition** becomes false,

the loop ends.
increment/ decrement: executed each time through the loop when condition is true.

Example Code

```
// Dim an LED using a PWM pin
int PWMpin = 10;  // LED in series with 470 ohm resistor on pin 10
void setup() {
  // no setup needed
}

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

while:

Description: A while loop will loop continuously, and infinitely, until the expression inside the parenthesis, () becomes false. Something must change the tested variable, or the while loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as testing a sensor.

Syntax

```
while (condition) {
  // statement(s)
}
```

Parameters
condition: a boolean expression that evaluates to true or false.

Example Code

```
var = 0;
while (var < 200) {
  // do something repetitive 200 times
  var++;
}
```

do...while:

Description: The do…while loop works in the same manner as the while loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run at least once.

Syntax

```
do {
  // statement block
} while (condition);
```

Parameters
condition: a boolean expression that evaluates to true or false.

Example Code
```
int x = 0;
do {
  delay(50);         // wait for sensors to stabilize
  x = readSensors();  // check the sensors
} while (x < 100);
```

switch...case:

Description: Like if statements, switch case controls the flow of programs by allowing programmers to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run.

The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.

Syntax
```
switch (var) {
  case label1:
    // statements
    break;
  case label2:
    // statements
    break;
  default:
    // statements
    break;
}
```

Parameters
var: a variable whose value to compare with various cases. Allowed data types: int, char. label1, label2: constants. Allowed data types: int, char.

Example Code
```
switch (var) {
  case 1:
    //do something when var equals 1
    break;
  case 2:
    //do something when var equals 2
    break;
  default:
    // if nothing else matches, do the default
```

```
  // default is optional
  break;
}
```

break:

Description: break is used to exit from a for, while or do…while loop, bypassing the normal loop condition. It is also used to exit from a switch case statement.

Example Code

In the following code, the control exits the for loop when the sensor value exceeds the threshold.

```
int threshold = 40;
for (int x = 0; x < 255; x++) {
 analogWrite(PWMpin, x);
 sens = analogRead(sensorPin);
 if (sens > threshold) {    // bail out on sensor detect
  x = 0;
  break;
 }
 delay(50);
}
```

continue:

Description: The continue statement skips the rest of the current iteration of a loop (for, while, or do…while). It continues by checking the conditional expression of the loop, and proceeding with any subsequent iterations.

Example Code

The following code writes the value of 0 to 255 to the PWMpin, but skips the values in the range of 41 to 119.

```
for (int x = 0; x <= 255; x ++) {
 if (x > 40 && x < 120) {  // create jump in values
  continue;
 }
 analogWrite(PWMpin, x);
 delay(50);
}
```

return:

Description: Terminate a function and return a value from a function to the calling function, if desired.

Syntax

```
return;
return value;
```

Parameters

value: Allowed data types: any variable or constant type.

Example Code
A function to compare a sensor input to a threshold

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

The return keyword is handy to test a section of code without having to "comment out" large sections of possibly buggy code.

goto:

Description: Transfers program flow to a labeled point in the program

Syntax

```
label:
goto label; // sends program flow to the label
```

Example Code

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

bailout:
Serial.print("Hi") .
```

Further Syntax:

#define

Description: #define is a useful C++ component that allows the programmer to give a name to a constant value before the program is compiled. Defined constants in arduino don't take up any program memory space on the chip. The compiler will replace references to these constants with the defined value at compile time.

Syntax

```
#define constantName value
```

Example Code: #define ledPin 3

#include:

Description: #include is used to include outside libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino.

Example Code
This example includes the Servo library so that its functions may be used to control a Servo motor.

```
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
void setup() {
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}
void loop() {
  for (int pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 180 degrees
    // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
  for (int pos = 180; pos >= 0; pos -= 1) { // goes from 180 degrees to 0 degrees
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
}
```

/* */(Block comment):
Description: Comments are lines in the program that are used to inform yourself or others about the way the program works. They are ignored by the compiler, and not exported to the processor, so they don't take up any space in the microcontroller's flash memory. Comments' only purpose is to help you understand (or remember), or to inform others about how your program works.

The beginning of a block comment or a multi-line comment is marked by the symbol /* and the symbol */ marks its end. This type of comment is called so as this can extend over more than one line; once the compiler reads the /* it ignores whatever follows until it encounters a */.

Example Code

```
/* This is a valid comment */
```

//(Singlelinecomment):
Description: A single line comment begins with // (two adjacent slashes). This comment ends automatically at the end of a line. Whatever follows // till the end of a line will be ignored by the compiler.

Example Code
There are two different ways of marking a line as a comment:

```
// pin 13 has an LED connected on most Arduino boards.
digitalWrite(led, HIGH);  // turn the LED on (HIGH is the voltage level)
```

; (Semicolon):

Description: Used to end a statement.

Example Code
int a = 13;

{ }(Curlybraces):
Description: Curly braces (also referred to as just "braces" or as "curly brackets"). An opening curly brace { must always be followed by a closing curly brace }. This is a condition that is often referred to as the braces being balanced.

Example Code:
The main uses of curly braces are listed in the examples below.

Functions
```
void myfunction(datatype argument) {
  // any statement(s)
}
```

Loops
```
while (boolean expression) {
  // any statement(s)
}
```

Arithmetic Operators:

% (remainder)
Description: Remainder operation calculates the remainder when one integer is divided by another. It is useful for keeping a variable within a particular range (e.g. the size of an array). The % (percent) symbol is used to carry out remainder operation.

Syntax
```
remainder = dividend % divisor;
```

Example Code
```
int x = 0;
x = 7 % 5;  // x now contains 2
x = 9 % 5;  // x now contains 4
```

*** (multiplication):**
Description: Multiplication is one of the four primary arithmetic operations. The operator * (asterisk) operates on two operands to produce the product.

Syntax
```
product = operand1 * operand2;
```

Example Code
```
int a = 5;
int b = 10;
```

```
int c = 0;
c = a * b;
```

+ (addition):

Description: Addition is one of the four primary arithmetic operations. The operator + (plus) operates on two operands to produce the sum.

Syntax

```
sum = operand1 + operand2;
```

Example Code

```
int a = 5;
int b = 10;
int c = 0;
c = a + b;
```

- (subtraction):

Description: Subtraction is one of the four primary arithmetic operations. The operator - (minus) operates on two operands to produce the difference of the second from the first.

Syntax

```
difference = operand1 - operand2;
```

Example Code

```
int a = 5;
int b = 10;
int c = 0;
c = b-a;
```

/ (division):

Description: Division is one of the four primary arithmetic operations. The operator / (slash) operates on two operands to produce the result.

Syntax

```
result = numerator / denominator;
```

Example Code

```
int a = 5;
int b = 10;
int c = 0;
c = b/a;
```

= (assignment operator):

Description: The single equal sign = in the C++ programming language is called the assignment operator. It has a different meaning than in algebra class where it indicated an equation or equality. The assignment operator tells the microcontroller to evaluate whatever value or expression is on the right side of the equal sign, and store it in the variable to the left of the equal sign.

Example Code
```
int sensVal;            // declare an integer variable named sensVal
sensVal = analogRead(0);  // store the (digitized) input voltage at analog pin 0 in SensVal
```

Comparison Operators

!= (not equal to)

Description: Compares the variable on the left with the value or variable on the right of the operator. Returns true when the two operands are not equal.

Syntax
```
x != y; // is false if x is equal to y and it is true if x is not equal to y
```

Example Code
```
if (x != y) { // tests if x is not equal to y
  // do something only if the comparison result is true
}
```

< (less than):

Description: Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is less (smaller) than the operand on the right.

Syntax
```
x < y; // is true if x is smaller than y and it is false if x is equal or bigger than y
```

Example Code
```
if (x < y) {  // tests if x is less (smaller) than y
  // do something only if the comparison result is true
}
```

<= (less than or equal to):

Description: Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is less (smaller) than or equal to the operand on the right.

Syntax
```
x ⇐ y; // is true if x is smaller than or equal to y and it is false if x is greater than y
```

Example Code
```
if (x <= y) { // tests if x is less (smaller) than or equal to y
  // do something only if the comparison result is true
}
```

== (equal to):

Description: Compares the variable on the left with the value or variable on the right of the operator. Returns true when the two operands are equal.

Syntax
```
x == y; // is true if x is equal to y and it is false if x is not equal to y
```

Example Code
```
if (x == y) { // tests if x is equal to y
  // do something only if the comparison result is true
}
```

> (greater than):
Description: Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is greater (bigger) than the operand on the right.

Syntax
```
x > y; // is true if x is bigger than y and it is false if x is equal or smaller than y
```

Example Code
```
if (x > y) {  // tests if x is greater (bigger) than y
  // do something only if the comparison result is true
}
```

>= (greater than or equal to)
Description: Compares the variable on the left with the value or variable on the right of the operator. Returns true when the operand on the left is greater (bigger) than or equal to the operand on the right.

Syntax
```
x >= y; // is true if x is bigger than or equal to y and it is false if x is smaller than y
```

Example Code
```
if (x >= y) { // tests if x is greater (bigger) than or equal to y
  // do something only if the comparison result is true
}
```

Boolean Operators

! (logical not)
Description: Logical NOT results in a true if the operand is false and vice versa.

Example Code
This operator can be used inside the condition of an if statement.

```
if (!x) { // if x is not true
  // statements
}
```

&& (logical and)
Description: Logical AND results in true only if both operands are true.

Example Code
This operator can be used inside the condition of an if statement.

```
if (digitalRead(2) == HIGH && digitalRead(3) == HIGH) { // if BOTH the switches read HIGH
  // statements
}
```

|| (logical or)

Description: Logical OR results in a true if either of the two operands is true.

Example Code

This operator can be used inside the condition of an if statement.

```
if (x > 0 || y > 0) { // if either x or y is greater than zero
  // statements
}
```

Pointer Access Operators

& (reference operator)

Description: Referencing is one of the features specifically for use with pointers. The ampersand operator & is used for this purpose. If x is a variable, then &x represents the address of the variable x.

Example Code

```
int *p;      // declare a pointer to an int data type
int i = 5;
int result = 0;
p = &i;      // now 'p' contains the address of 'i'
result = *p;  // 'result' gets the value at the address pointed by 'p'
          // i.e., it gets the value of 'i' which is 5
```

*** (dereference operator)**

Description: Dereferencing is one of the features specifically for use with pointers. The asterisk operator * is used for this purpose. If p is a pointer, then *p represents the value contained in the address pointed by p.

Example Code

```
int *p;      // declare a pointer to an int data type
int i = 5;
int result = 0;
p = &i;      // now 'p' contains the address of 'i'
result = *p;  // 'result' gets the value at the address pointed by 'p'
          // i.e., it gets the value of 'i' which is 5
```

Bitwise Operators

& (bitwise and)

Description: The bitwise AND operator in C++ is a single ampersand &, used between two other integer expressions. Bitwise AND operates on each bit position of the surrounding expressions independently, according to this rule: if both input bits are 1, the resulting output is 1, otherwise the output is 0.

In Arduino, the type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur.

Example Code
In a code fragment like:

```
int a =  92;   // in binary: 0000000001011100
int b = 101;   // in binary: 0000000001100101
int c = a & b;  // result:   0000000001000100, or 68 in decimal
```

<< (bitshift left)
Description: The left shift operator << causes the bits of the left operand to be shifted left by the number of positions specified by the right operand.

Syntax
```
variable << number_of_bits;
```

Example Code
```
int a = 5;     // binary: 0000000000000101
int b = a << 3; // binary: 0000000000101000, or 40 in decimal
```

>> (bitshift right):
Description: The right shift operator >> causes the bits of the left operand to be shifted right by the number of positions specified by the right operand.

Syntax
```
variable >> number_of_bits;
```

Example Code
```
int a = 40;     // binary: 0000000000101000
int b = a >> 3; // binary: 0000000000000101, or 5 in decimal
```

^ (bitwise xor)
Description: There is a somewhat unusual operator in C++ called bitwise EXCLUSIVE OR, also known as bitwise XOR. The bitwise XOR operator is written using the caret symbol ^. A bitwise XOR operation results in a 1 only if the input bits are different, else it results in a 0.

Example Code
```
int x = 12;     // binary: 1100
int y = 10;     // binary: 1010
int z = x ^ y;  // binary: 0110, or decimal 6
```

| (bitwise or)
Description: The bitwise OR operator in C++ is the vertical bar symbol, |. Like the & operator, | operates independently each bit in its two surrounding integer expressions, but what it does is different (of course). The bitwise OR of two bits is 1 if either or both of the input bits is 1, otherwise it is 0.

Example Code
int a = 92; // in binary: 0000000001011100
int b = 101; // in binary: 0000000001100101
int c = a | b; // result: 0000000001111101, or 125 in decimal

~ (bitwise not)
Description: The bitwise NOT operator in C++ is the tilde character ~. Unlike & and |, the bitwise NOT operator is applied to a single operand to its right. Bitwise NOT changes each bit to its opposite: 0 becomes 1, and 1 becomes 0.

Example Code
int a = 103; // binary: 0000000001100111
int b = ~a; // binary: 1111111110011000 = -104

Compound Operators

%= (compound remainder)
Description: This is a convenient shorthand to calculate the remainder when one integer is divided by another and assign it back to the variable the calculation was done on.

Syntax
x %= divisor; // equivalent to the expression x = x % divisor;

Example Code
int x = 7;
x %= 5; // x now contains 2

&= (compound bitwise and)
Description: The compound bitwise AND operator &= is often used with a variable and a constant to force particular bits in a variable to the LOW state (to 0). This is often referred to in programming guides as "clearing" or "resetting" bits.

Syntax
x &= y; // equivalent to x = x & y;

Example Code
Bits that are "bitwise ANDed" with 0 are cleared to 0 so, if myByte is a byte variable,

myByte & B00000000 = 0;
Bits that are "bitwise ANDed" with 1 are unchanged so,

myByte & B11111111 = myByte;

***= (compound multiplication)**
Description: This is a convenient shorthand to perform multiplication of a variable with another constant or variable.

Syntax ux *= y; // equivalent to the expression x = x * y;

Example Code
x = 2;
x *= 2; // x now contains 4

++ (increment)
Description: Increments the value of a variable by 1.

Syntax
x; // increment x by one and returns the old value of x` + `x; // increment x by one and returns the new value of x

Example Code
x = 2;
y = ++x; // x now contains 3, y contains 3
y = x++; // x contains 4, but y still contains 3

+= (compound addition)
Description: This is a convenient shorthand to perform addition on a variable with another constant or variable.

Syntax
x += y; // equivalent to the expression x = x + y;

Example Code
x = 2;
x += 4; // x now contains 6

-- (decrement)
Description: Decrements the value of a variable by 1.

Syntax
x--; // decrement x by one and returns the old value of x
--x; // decrement x by one and returns the new value of x

Example Code
x = 2;
y = --x; // x now contains 1, y contains 1
y = x--; // x contains 0, but y still contains 1

-= (compound subtraction)
Description: This is a convenient shorthand to perform subtraction of a constant or a variable from a variable.

Syntax
x -= y; // equivalent to the expression x = x - y;

Example Code
x = 20;
x -= 2; // x now contains 18

/= (compound division)
Description: This is a convenient shorthand to perform division of a variable with another constant or variable.

Syntax
x /= y; // equivalent to the expression x = x / y;

Example Code
x = 2;
x /= 2; // x now contains 1

^= (compound bitwise xor)
Description: The compound bitwise XOR operator ^= is often used with a variable and a constant to toggle (invert) particular bits in a variable.

Syntax
x ^= y; // equivalent to x = x ^ y;

Example Code
Bits that are "bitwise XORed" with 0 are left unchanged. So if myByte is a byte variable,

myByte ^ B00000000 = myByte;
Bits that are "bitwise XORed" with 1 are toggled so:

myByte ^ B11111111 = ~myByte;

|= (compound bitwise or)
Description: The compound bitwise OR operator |= is often used with a variable and a constant to "set" (set to 1) particular bits in a variable.

Syntax
x |= y; // equivalent to x = x | y;

Example Code
Bits that are "bitwise ORed" with 0 are unchanged, so if myByte is a byte variable,

myByte | B00000000 = myByte;
Bits that are "bitwise ORed" with 1 are set to 1 so:

myByte | B11111111 = B11111111;

VARIABLES

Arduino data types and constants.

Constants

- **Floating Point Constants**
- **Integer Constants**
- **HIGH | LOW**
- **INPUT | OUTPUT | INPUT_PULLUP**
- **LED_BUILTIN**
- **true | false**

Floating Point Constants

Description: Similar to integer constants, floating point constants are used to make code more readable. Floating point constants are swapped at compile time for the value to which the expression evaluates.

Floating point constants can also be expressed in a variety of scientific notation. 'E' and 'e' are both accepted as valid exponent indicators.

FLOATING-POINT CONSTANT	EVALUATES TO:	ALSO EVALUATES TO:
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	0.000000000067

Example Code

```
n = 0.005;  // 0.005 is a floating point constant
```

Integer Constants

Description: Integer constants are numbers that are used directly in a sketch, like 123. Normally, integer constants are treated as base 10 (decimal) integers, but special notation (formatters) may be used to enter numbers in other bases.

BASE	EXAMPLE	FORMATTER	COMMENT
10 (decimal)	123	none	
2 (binary)	B1111011	leading 'B'	only works with 8 bit values (0 to 255) characters 0&1 valid
8 (octal)	0173	leading "0"	characters 0-7 valid

BASE	EXAMPLE	FORMATTER	COMMENT
16 (hexadecimal)	0x7B	leading "0x"	characters 0-9, A-F, a-f valid

constants

Description

Constants are predefined expressions in the Arduino language. They are used to make the programs easier to read. We classify constants in groups:

Defining Logical Levels: true and false (Boolean Constants)

There are two constants used to represent truth and falsity in the Arduino language: true, and false.

false

false is the easier of the two to define. false is defined as 0 (zero).

true

true is often said to be defined as 1, which is correct, but true has a wider definition. Any integer which is non-zero is true, in a Boolean sense. So -1, 2 and -200 are all defined as true, too, in a Boolean sense.

Defining Pin Levels: HIGH and LOW

When reading or writing to a digital pin there are only two possible values a pin can take/be-set-to: HIGH and LOW.

HIGH

The meaning of HIGH (in reference to a pin) is somewhat different depending on whether a pin is set to an INPUT or OUTPUT. When a pin is configured as an INPUT with pinMode(), and read with digitalRead(), the Arduino (ATmega) will report HIGH if:

- a voltage greater than 3.0V is present at the pin (5V boards)
- a voltage greater than 2.0V volts is present at the pin (3.3V boards)

When a pin is configured to OUTPUT with pinMode(), and set to HIGH with digitalWrite(), the pin is at:

- 5 volts (5V boards)
- volts (3.3V boards)

LOW

The meaning of LOW also has a different meaning depending on whether a pin is set to INPUT or OUTPUT. When a pin is configured as an INPUT with pinMode(), and read with digitalRead(), the Arduino (ATmega) will report LOW if:

- a voltage less than 1.5V is present at the pin (5V boards)
- a voltage less than 1.0V (Approx) is present at the pin (3.3V boards)

When a pin is configured to OUTPUT with pinMode(), and set to LOW with digitalWrite(), the pin is at 0 volts (both 5V and 3.3V boards).

Defining Digital Pins modes: INPUT, INPUT_PULLUP, and OUTPUT
Digital pins can be used as INPUT, INPUT_PULLUP, or OUTPUT. Changing a pin with pinMode() changes the electrical behavior of the pin.

Pins Configured as INPUT
Arduino (ATmega) pins configured as INPUT with pinMode() are said to be in a high-impedance state. If you have your pin configured as an INPUT, and are reading a switch, when the switch is in the open state the input pin will be "floating", resulting in unpredictable results. In order to assure a proper reading when the switch is open, a pull-up or pull-down resistor must be used.

- If a pull-down resistor is used, the input pin will be LOW when the switch is open and HIGH when the switch is closed.
- If a pull-up resistor is used, the input pin will be HIGH when the switch is open and LOW when the switch is closed.

Pins Configured as INPUT_PULLUP
The ATmega microcontroller on the Arduino has internal pull-up resistors (resistors that connect to power internally) that you can access. If you prefer to use these instead of external pull-up resistors, you can use the INPUT_PULLUP argument in pinMode().

Pins Configured as OUTPUT
Pins configured as OUTPUT with pinMode() are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits.

Defining built-ins: LED_BUILTIN
Most Arduino boards have a pin connected to an on-board LED in series with a resistor. The constant LED_BUILTIN is the number of the pin to which the on-board LED is connected. Most boards have this LED connected to digital pin 13.

Conversion

- (unsigned int)
- (unsigned long)
- byte()
- char()
- float()
- int()
- long()
- word()

(unsigned int)
Description
Converts a value to the unsigned int data type.

Syntax
(unsigned int)x

Parameters
x: a value of any type

Returns
unsigned int

(unsigned long)
Description
Converts a value to the unsigned long data type.

Syntax
(unsigned long)x

Parameters
x: a value of any type

Returns
unsigned long

byte()
Description
Converts a value to the byte data type.

Syntax
byte(x)
(byte)x (C-style type conversion)

Parameters
x: a value. Allowed data types: any type.

Returns
Data type: byte.

char()
Description
Converts a value to the char data type.

Syntax
char(x)
(char)x (C-style type conversion)

Parameters
x: a value. Allowed data types: any type.

Returns
Data type: char.

int()

Description

Converts a value to the int data type.

Syntax

int(x)
(int)x (C-style type conversion)

Parameters

x: a value. Allowed data types: any type.

Returns

Data type: int.

float()

Description

Converts a value to the float data type.

Syntax

float(x)
(float)x (C-style type conversion)

Parameters

x: a value. Allowed data types: any type.

Returns

Data type: float.

long()

Description

Converts a value to the long data type.

Syntax

long(x)
(long)x (C-style type conversion)

Parameters

x: a value. Allowed data types: any type.

Returns

Data type: long.

word()

Description

Converts a value to the word data type.

Syntax

word(x)
word(h, l)
(word)x (C-style type conversion)

Parameters
x: a value. Allowed data types: any type.
h: the high-order (leftmost) byte of the word.
l: the low-order (rightmost) byte of the word.

Returns
Data type: word.

Data Types

String()

Description
Constructs an instance of the String class. There are multiple versions that construct Strings from different data types (i.e. format them as sequences of characters), including:

- a constant string of characters, in double quotes (i.e. a char array)
- a single constant character, in single quotes
- another instance of the String object
- a constant integer or long integer
- a constant integer or long integer, using a specified base
- an integer or long integer variable
- an integer or long integer variable, using a specified base
- a float or double, using a specified decimal places

Example Code

All of the following are valid declarations for Strings.

```
String stringOne = "Hello String";                 // using a constant String
String stringOne = String('a');                    // converting a constant char into a String
String stringTwo = String("This is a string");     // converting a constant string into a String object
String stringOne = String(stringTwo + " with more");  // concatenating two strings
String stringOne = String(13);                     // using a constant integer
String stringOne = String(analogRead(0), DEC);     // using an int and a base
String stringOne = String(45, HEX);                // using an int and a base (hexadecimal)
String stringOne = String(255, BIN);               // using an int and a base (binary)
String stringOne = String(millis(), DEC);          // using a long and a base
String stringOne = String(5.698, 3);               // using a float and the decimal places
```

array

Description
An array is a collection of variables that are accessed with an index number. Arrays in the C++ programming language Arduino sketches are written in can be complicated, but using simple arrays is relatively straightforward.

All of the methods below are valid ways to create (declare) an array.

- int myInts[6];

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

bool

Description

A bool holds one of two values, true or false. (Each bool variable occupies one byte of memory.)

Syntax

bool var = val;

Parameters

var: variable name.
val: the value to assign to that variable.

boolean

Description

boolean is a non-standard type alias for bool defined by Arduino. It's recommended to instead use the standard type bool, which is identical.

byte

Description

A byte stores an 8-bit unsigned number, from 0 to 255.

Syntax

byte var = val;

Parameters

var: variable name.
val: the value to assign to that variable.

char

Description

A data type used to store a character value. Character literals are written in single quotes, like this: 'A' (for multiple characters - strings - use double quotes: "ABC").

The size of the char datatype is at least 8 bits. It's recommended to only use char for storing characters. For an unsigned, one-byte (8 bit) data type, use the byte data type.

Syntax

char var = val;

Parameters

var: variable name.
val: the value to assign to that variable.

Example Code

```
char myChar = 'A';
char myChar = 65; // both are equivalent
```

double

Description

Double precision floating point number. On the Uno and other ATMEGA based boards, this occupies 4 bytes. That is, the double implementation is exactly the same as the float, with no gain in precision.

On the Arduino Due, doubles have 8-byte (64 bit) precision.

Syntax

```
double var = val;
```

Parameters

var: variable name.
val: the value to assign to that variable.

float

Description

Datatype for floating-point numbers, a number that has a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have greater resolution than integers. Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

Syntax

```
float var = val;
```

Parameters

var: variable name.
val: the value you assign to that variable.

Example Code

```
float myfloat;
float sensorCalbrate = 1.117;

int x;
int y;
float z;

x = 1;
y = x / 2;        // y now contains 0, ints can't hold fractions
z = (float)x / 2.0; // z now contains .5 (you have to use 2.0, not 2)
```

int

Description

Integers are your primary data-type for number storage.

On the Arduino Uno (and other ATmega based boards) an int stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

Syntax
int var = val;

Parameters
var: variable name.
val: the value you assign to that variable.

long
Description
Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647.

If doing math with integers, at least one of the numbers must be followed by an L, forcing it to be a long. See the Integer Constants page for details.

Syntax
long var = val;

Parameters
var: variable name.
val: the value assigned to the variable.

Example Code
long speedOfLight = 186000L; // see the Integer Constants page for explanation of the 'L'

short
Description
A short is a 16-bit data-type.

On all Arduinos (ATMega and ARM based) a short stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

Syntax
short var = val;

Parameters
var: variable name.
val: the value you assign to that variable.

Example Code
short ledPin = 13

size_t
Description

size_t is a data type capable of representing the size of any object in bytes. Examples of the use of size_t are the return type of sizeof() and Serial.print().

Syntax
size_t var = val;

Parameters
var: variable name.
val: the value to assign to that variable.

unsigned char
Description
- An unsigned data type that occupies 1 byte of memory. Same as the byte datatype.
- The unsigned char datatype encodes numbers from 0 to 255.
- For consistency of Arduino programming style, the byte data type is to be preferred.

Syntax
unsigned char var = val;

Parameters
var: variable name.
val: the value to assign to that variable.

Example Code
unsigned char myChar = 240;

unsigned int
Description
On the Uno and other ATMEGA based boards, unsigned ints (unsigned integers) are the same as ints in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65,535 ((2^16) - 1).

Syntax
unsigned int var = val;

Parameters
var: variable name.
val: the value you assign to that variable.

Example Code
unsigned int ledPin = 13;

unsigned long
Description
Unsigned long variables are extended size variables for number storage, and store 32 bits (4 bytes). Unlike standard longs unsigned longs won't store negative numbers, making their range from 0 to 4,294,967,295 (2^32 - 1).

Syntax
unsigned long var = val;

Parameters
var: variable name.
val: the value you assign to that variable.

void
Description
The void keyword is used only in function declarations. It indicates that the function is expected to return no information to the function from which it was called.

Example Code
The code shows how to use void.

```
// actions are performed in the functions "setup" and "loop"
// but  no information is reported to the larger program

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

word
Description
A word can store an unsigned number of at least 16 bits (from 0 to 65535).

Syntax
word var = val;

Parameters
var: variable name.
val: the value to assign to that variable.

Example Code
word w = 10000;

Variable Scope & Qualifiers

- const
- scope
- static
- volatile

const
Description

The const keyword stands for constant. It is a variable qualifier that modifies the behavior of the variable, making a variable "read-only". This means that the variable can be used just as any other variable of its type, but its value cannot be changed. You will get a compiler error if you try to assign a value to a const variable.

Example Code

```
const float pi = 3.14;
float x;
// ....
x = pi * 2; // it's fine to use consts in math
pi = 7;     // illegal - you can't write to (modify) a constant
```

scope

Description

Variables in the C++ programming language, which Arduino uses, have a property called scope. This is in contrast to early versions of languages such as BASIC where every variable is a global variable.

A global variable is one that can be seen by every function in a program. Local variables are only visible to the function in which they are declared. In the Arduino environment, any variable declared outside of a function (e.g. setup(), loop(), etc.), is a global variable.

Example Code

```
int gPWMval;  // any function will see this variable

void setup() {
  // ...
}

void loop() {
  int i;    // "i" is only "visible" inside of "loop"
  float f;  // "f" is only "visible" inside of "loop"
  // ...

  for (int j = 0; j < 100; j++) {
    // variable j can only be accessed inside the for-loop brackets
  }
}
```

static

Description

The static keyword is used to create variables that are visible to only one function. However unlike local variables that get created and destroyed every time a function is called, static variables persist beyond the function call, preserving their data between function calls.

Variables declared as static will only be created and initialized the first time a function is called.

Example Code

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

int thisTime;

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

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

int randomWalk(int moveSize) {
  static int place; // variable to store value in random walk - declared static so that it stores
  // values in between function calls, but no other functions can change its value

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange) {                    // check lower and upper limits
    place = randomWalkLowRange + (randomWalkLowRange - place);     // reflect number back in positive direction
  }
  else if (place > randomWalkHighRange) {
    place = randomWalkHighRange - (place - randomWalkHighRange);  // reflect number back in negative direction
  }

  return place;
}
```

volatile

Description

volatile is a keyword known as a variable qualifier, it is usually used before the datatype of a variable, to modify the way in which the compiler and subsequent program treat the variable.

Specifically, it directs the compiler to load the variable from RAM and not from a storage register, which is a temporary memory location where program variables are stored and manipulated. Under certain conditions, the value for a variable stored in registers can be inaccurate.

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called an interrupt service routine.

Utilities

- PROGMEM
- sizeof()

PROGMEM

Description

Store data in flash (program) memory instead of SRAM. There's a description of the various types of memory available on an Arduino board.

The PROGMEM keyword is a variable modifier, it should be used only with the datatypes defined in pgmspace.h. It tells the compiler "put this information into flash memory", instead of into SRAM, where it would normally go.

PROGMEM is part of the pgmspace.h library. It is included automatically in modern versions of the IDE.

Syntax

const dataType variableName[] PROGMEM = {data0, data1, data3…};

Parameters

dataType: Allowed data types: any variable type.
variableName: the name for your array of data.

Example Code

The following code fragments illustrate how to read and write unsigned chars (bytes) and ints (2 bytes) to PROGMEM.

```
// save some unsigned ints
const PROGMEM uint16_t charSet[] = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM = {"I AM PREDATOR,  UNSEEN COMBATANT. CREATED BY THE ECE DEPARTMENT"};
```

sizeof()

Description

The sizeof operator returns the number of bytes in a variable type, or the number of bytes occupied by an array.

Syntax
sizeof(variable)

Parameters
variable: The thing to get the size of. Allowed data types: any variable type or array (e.g. int, float, byte).

Returns
The number of bytes in a variable or bytes occupied in an array. Data type: size_t.

Example Code
The sizeof operator is useful for dealing with arrays (such as strings) where it is convenient to be able to change the size of the array without breaking other parts of the program.

This program prints out a text string one character at a time. Try changing the text phrase.

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

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

void loop() {
  for (byte i = 0; i < sizeof(myStr) - 1; i++) {
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000);  // slow down the program
}
```

FUNCTIONS

For controlling the Arduino board and performing computations.

Digital I/O

- digitalRead()
- digitalWrite()
- pinMode()

digitalRead()

Description

Reads the value from a specified digital pin, either HIGH or LOW.

Syntax

digitalRead(pin)

Parameters

pin: the Arduino pin number you want to read

Returns

HIGH or LOW

Example Code

Sets pin 13 to the same value as pin 7, declared as an input.

```
int ledPin = 13;  // LED connected to digital pin 13
int inPin = 7;    // pushbutton connected to digital pin 7
int val = 0;      // variable to store the read value

void setup() {
  pinMode(ledPin, OUTPUT);  // sets the digital pin 13 as output
  pinMode(inPin, INPUT);    // sets the digital pin 7 as input
}

void loop() {
  val = digitalRead(inPin);   // read the input pin
  digitalWrite(ledPin, val);  // sets the LED to the button's value
}
```

digitalWrite()

Description

Write a HIGH or a LOW value to a digital pin.

If the pin has been configured as an OUTPUT with pinMode(), its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW.

Syntax

digitalWrite(pin, value)

Parameters
pin: the Arduino pin number.
value: HIGH or LOW.

Returns
Nothing

Example Code
The code makes the digital pin 13 an OUTPUT and toggles it by alternating between HIGH and LOW at one second pace.

```
void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as output
}

void loop() {
  digitalWrite(13, HIGH); // sets the digital pin 13 on
  delay(1000);            // waits for a second
  digitalWrite(13, LOW);  // sets the digital pin 13 off
  delay(1000);            // waits for a second
}
```

pinMode()
Description
Configures the specified pin to behave either as an input or an output. See the Digital Pins page for details on the functionality of the pins.

As of Arduino 1.0.1, it is possible to enable the internal pullup resistors with the mode INPUT_PULLUP. Additionally, the INPUT mode explicitly disables the internal pullups.

Syntax
pinMode(pin, mode)

Parameters
pin: the Arduino pin number to set the mode of.
mode: INPUT, OUTPUT, or INPUT_PULLUP. See the Digital Pins page for a more complete description of the functionality.

Returns
Nothing

Example Code
The code makes the digital pin 13 OUTPUT and Toggles it HIGH and LOW

```
void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as output
}
```

```
void loop() {
  digitalWrite(13, HIGH); // sets the digital pin 13 on
  delay(1000);            // waits for a second
  digitalWrite(13, LOW);  // sets the digital pin 13 off
  delay(1000);            // waits for a second
}
```

Analog I/O

- analogRead()
- analogReference()
- analogWrite()

analogRead()

Description

Reads the value from the specified analog pin. Arduino boards contain a multichannel, 10-bit analog to digital converter. This means that it will map input voltages between 0 and the operating voltage(5V or 3.3V) into integer values between 0 and 1023. On an Arduino UNO, for example, this yields a resolution between readings of: 5 volts / 1024 units or, 0.0049 volts (4.9 mV) per unit.

Syntax

analogRead(pin)

Parameters

pin: the name of the analog input pin to read from (A0 to A5 on most boards, A0 to A6 on MKR boards, A0 to A7 on the Mini and Nano, A0 to A15 on the Mega).

Returns

The analog reading on the pin. Although it is limited to the resolution of the analog to digital converter (0-1023 for 10 bits or 0-4095 for 12 bits). Data type: int.

Example Code

The code reads the voltage on analogPin and displays it.

```
int analogPin = A3; // potentiometer wiper (middle terminal) connected to analog pin 3
                    // outside leads to ground and +5V
int val = 0;  // variable to store the value read

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

void loop() {
  val = analogRead(analogPin);  // read the input pin
  Serial.println(val);          // debug value
}
```

analogWrite()

Description

Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to analogWrite(), the pin will generate a steady rectangular wave of the specified duty cycle until the next call to analogWrite() (or a call to digitalRead() or digitalWrite()) on the same pin.

Syntax

analogWrite(pin, value)

Parameters

pin: the Arduino pin to write to. Allowed data types: int.
value: the duty cycle: between 0 (always off) and 255 (always on). Allowed data types: int.

Returns

Nothing

Example Code

Sets the output to the LED proportional to the value read from the potentiometer.

```
int ledPin = 9;      // LED connected to digital pin 9
int analogPin = 3;   // potentiometer connected to analog pin 3
int val = 0;         // variable to store the read value

void setup() {
  pinMode(ledPin, OUTPUT);  // sets the pin as output
}

void loop() {
  val = analogRead(analogPin);  // read the input pin
  analogWrite(ledPin, val / 4); // analogRead values go from 0 to 1023, analogWrite values from 0 to 255
}
```

Advanced I/O

- noTone()
- pulseIn()
- pulseInLong()
- shiftIn()
- shiftOut()
- tone()

tone()

Description

Generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to noTone(). The pin can be connected to a piezo buzzer or other speaker to play tones.

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

Parameters
pin: the Arduino pin on which to generate the tone.
frequency: the frequency of the tone in hertz. Allowed data types: unsigned int.
duration: the duration of the tone in milliseconds (optional). Allowed data types: unsigned long.

Returns
Nothing

noTone()
Description
Stops the generation of a square wave triggered by tone(). Has no effect if no tone is being generated.

Syntax
noTone(pin)

Parameters
pin: the Arduino pin on which to stop generating the tone

Returns
Nothing

pulseIn()
Description
Reads a pulse (either HIGH or LOW) on a pin. For example, if value is HIGH, pulseIn() waits for the pin to go from LOW to HIGH, starts timing, then waits for the pin to go LOW and stops timing. Returns the length of the pulse in microseconds or gives up and returns 0 if no complete pulse was received within the timeout.

The timing of this function has been determined empirically and will probably show errors in longer pulses. Works on pulses from 10 microseconds to 3 minutes in length.

Syntax
pulseIn(pin, value)
pulseIn(pin, value, timeout)

Parameters
pin: the number of the Arduino pin on which you want to read the pulse. Allowed data types: int.
value: type of pulse to read: either HIGH or LOW. Allowed data types: int.
timeout (optional): the number of microseconds to wait for the pulse to start; default is one second. Allowed data types: unsigned long.

Returns
The length of the pulse (in microseconds) or 0 if no pulse started before the timeout. Data type: unsigned long.

Example Code
The example prints the time duration of a pulse on pin 7.

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

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

void loop() {
  duration = pulseIn(pin, HIGH);
  Serial.println(duration);
}
```

pulseInLong()
Description
pulseInLong() is an alternative to pulseIn() which is better at handling long pulse and interrupt affected scenarios.

Reads a pulse (either HIGH or LOW) on a pin. For example, if value is HIGH, pulseInLong() waits for the pin to go from LOW to HIGH, starts timing, then waits for the pin to go LOW and stops timing. Returns the length of the pulse in microseconds or gives up and returns 0 if no complete pulse was received within the timeout.

The timing of this function has been determined empirically and will probably show errors in shorter pulses. Works on pulses from 10 microseconds to 3 minutes in length. This routine can be used only if interrupts are activated. Furthermore the highest resolution is obtained with large intervals.

Syntax
pulseInLong(pin, value)
pulseInLong(pin, value, timeout)

Parameters
pin: the number of the Arduino pin on which you want to read the pulse. Allowed data types: int.
value: type of pulse to read: either HIGH or LOW. Allowed data types: int.
timeout (optional): the number of microseconds to wait for the pulse to start; default is one second. Allowed data types: unsigned long.

Returns
The length of the pulse (in microseconds) or 0 if no pulse started before the timeout. Data type: unsigned long.

Example Code
The example prints the time duration of a pulse on pin 7.

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

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

void loop() {
  duration = pulseInLong(pin, HIGH);
  Serial.println(duration);
}
```

shiftIn()
Description
Shifts in a byte of data one bit at a time. Starts from either the most (i.e. the leftmost) or least (rightmost) significant bit. For each bit, the clock pin is pulled high, the next bit is read from the data line, and then the clock pin is taken low.

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

Parameters
dataPin: the pin on which to input each bit. Allowed data types: int.
clockPin: the pin to toggle to signal a read from dataPin.
bitOrder: which order to shift in the bits; either MSBFIRST or LSBFIRST. (Most Significant Bit First, or, Least Significant Bit First).

Returns
The value read. Data type: byte.

shiftOut()
Description
Shifts out a byte of data one bit at a time. Starts from either the most (i.e. the leftmost) or least (rightmost) significant bit. Each bit is written in turn to a data pin, after which a clock pin is pulsed (taken high, then low) to indicate that the bit is available.

Syntax
shiftOut(dataPin, clockPin, bitOrder, value)

Parameters
dataPin: the pin on which to output each bit. Allowed data types: int.

clockPin: the pin to toggle once the dataPin has been set to the correct value. Allowed data types: int.
bitOrder: which order to shift out the bits; either MSBFIRST or LSBFIRST. (Most Significant Bit First, or, Least Significant Bit First).
value: the data to shift out. Allowed data types: byte.

Returns
Nothing

Time

- delay()
- delayMicroseconds()
- micros()
- millis()

delay()
Description
Pauses the program for the amount of time (in milliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Syntax
delay(ms)

Parameters
ms: the number of milliseconds to pause. Allowed data types: unsigned long.

Returns
Nothing

Example Code
The code pauses the program for one second before toggling the output pin.

```
int ledPin = 13;          // LED connected to digital pin 13

void setup() {
  pinMode(ledPin, OUTPUT);    // sets the digital pin as output
}

void loop() {
  digitalWrite(ledPin, HIGH); // sets the LED on
  delay(1000);              // waits for a second
  digitalWrite(ledPin, LOW);  // sets the LED off
  delay(1000);              // waits for a second
}
```

delayMicroseconds()

Description

Pauses the program for the amount of time (in microseconds) specified by the parameter. There are a thousand microseconds in a millisecond and a million microseconds in a second.

Currently, the largest value that will produce an accurate delay is 16383. This could change in future Arduino releases. For delays longer than a few thousand microseconds, you should use delay() instead.

Syntax

delayMicroseconds(us)

Parameters

us: the number of microseconds to pause. Allowed data types: unsigned int.

Returns

Nothing

Example Code

The code configures pin number 8 to work as an output pin. It sends a train of pulses of approximately 100 microseconds period. The approximation is due to execution of the other instructions in the code.

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

void setup() {
  pinMode(outPin, OUTPUT);    // sets the digital pin as output
}

void loop() {
  digitalWrite(outPin, HIGH); // sets the pin on
  delayMicroseconds(50);      // pauses for 50 microseconds
  digitalWrite(outPin, LOW);  // sets the pin off
  delayMicroseconds(50);      // pauses for 50 microseconds
}
```

micros()

Description

Returns the number of microseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 70 minutes.

Syntax

time = micros()

Parameters

None

Returns
Returns the number of microseconds since the Arduino board began running the current program. Data type: unsigned long.

Example Code
The code returns the number of microseconds since the Arduino board began.

```
unsigned long time;

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

  Serial.println(time); //prints time since program started
  delay(1000);          // wait a second so as not to send massive amounts of data
}
```

millis()
Description
Returns the number of milliseconds passed since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

Syntax
time = millis()

Parameters
None

Returns
Number of milliseconds passed since the program started. Data type: unsigned long.

Example Code
This example code prints on the serial port the number of milliseconds passed since the Arduino board started running the code itself.

```
unsigned long time;

void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Time: ");
  time = millis();
  Serial.println(time); //prints time since program started
```

```
  delay(1000);        // wait a second so as not to send massive amounts of data
}
```

Math

- abs()
- constrain()
- map()
- max()
- min()
- pow()
- sq()
- sqrt()

abs()

Description

Calculates the absolute value of a number.

Syntax

abs(x)

Parameters

x: the number

Returns

x: if x is greater than or equal to 0.
-x: if x is less than 0.

constrain()

Description

Constrains a number to be within a range.

Syntax

constrain(x, a, b)

Parameters

x: the number to constrain Allowed data types: all data types.
a: the lower end of the range. Allowed data types: all data types.
b: the upper end of the range. Allowed data types: all data types.

Returns

x: if x is between a and b.
a: if x is less than a.
b: if x is greater than b.

Example Code

The code limits the sensor values to between 10 to 150.

```
sensVal = constrain(sensVal, 10, 150);  // limits range of sensor values to between 10 and 150
```

map()

Description

Re-maps a number from one range to another. That is, a value of fromLow would get mapped to toLow, a value of fromHigh to toHigh, values in-between to values in-between, etc.

Does not constrain values to within the range, because out-of-range values are sometimes intended and useful. The constrain() function may be used either before or after this function, if limits to the ranges are desired.

Syntax

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

Parameters

value: the number to map.
fromLow: the lower bound of the value's current range.
fromHigh: the upper bound of the value's current range.
toLow: the lower bound of the value's target range.
toHigh: the upper bound of the value's target range.

Returns

The mapped value.

Example Code

```
/* Map an analog value to 8 bits (0 to 255) */
void setup() {}

void loop() {
  int val = analogRead(A0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

max()

Description

Calculates the maximum of two numbers.

Syntax

max(x, y)

Parameters

x: the first number. Allowed data types: any data type.
y: the second number. Allowed data types: any data type.

Returns

The larger of the two parameter values.

Example Code

The code ensures that sensVal is at least 20.

sensVal = max(sensVal, 20); // assigns sensVal to the larger of sensVal or 20

min()

Description

Calculates the minimum of two numbers.

Syntax

min(x, y)

Parameters

x: the first number. Allowed data types: any data type.
y: the second number. Allowed data types: any data type.

Returns

The smaller of the two numbers.

Example Code

The code ensures that it never gets above 100.

sensVal = min(sensVal, 100); // assigns sensVal to the smaller of sensVal or 100

pow()

Description

Calculates the value of a number raised to a power. pow() can be used to raise a number to a fractional power. This is useful for generating exponential mapping of values or curves.

Syntax

pow(base, exponent)

Parameters

base: the number. Allowed data types: float.
exponent: the power to which the base is raised. Allowed data types: float.

Returns

The result of the exponentiation. Data type: double.

Example Code

Calculate the value of x raised to the power of y:

z = pow(x, y);

sq()

Description

Calculates the square of a number: the number multiplied by itself.

Syntax
sq(x)

Parameters
x: the number. Allowed data types: any data type.

Returns
The square of the number. Data type: double.

Example code

```
int input = Serial.parseInt();  // keep other operations outside the sq function
int inputSquared = sq(input);
```

sqrt()
Description
Calculates the square root of a number.

Syntax
sqrt(x)

Parameters
x: the number. Allowed data types: any data type.

Returns
The number's square root. Data type: double.

Trigonometry

- cos()
- sin()
- tan()

cos()
Description
Calculates the cosine of an angle (in radians). The result will be between -1 and 1.

Syntax
cos(rad)

Parameters
rad: The angle in radians. Allowed data types: float.

Returns
The cos of the angle. Data type: double.

sin()

Description

Calculates the sine of an angle (in radians). The result will be between -1 and 1.

Syntax

sin(rad)

Parameters

rad: The angle in radians. Allowed data types: float.

Returns

The sine of the angle. Data type: double.

tan()

Description

Calculates the tangent of an angle (in radians). The result will be between negative infinity and infinity.

Syntax

tan(rad)

Parameters

rad: The angle in radians. Allowed data types: float.

Returns

The tangent of the angle. Data type: double.

Characters

- isAlpha()
- isAlphaNumeric()
- isAscii()
- isControl()
- isDigit()
- isGraph()
- isHexadecimalDigit()
- isLowerCase()
- isPrintable()
- isPunct()
- isSpace()
- isUpperCase()
- isWhitespace()

isAlpha()

Description

Analyse if a char is alpha (that is a letter). Returns true if thisChar contains a letter.

Syntax
isAlpha(thisChar)

Parameters
thisChar: variable. Allowed data types: char.

Returns
true: if thisChar is alpha.

Example Code
```
if (isAlpha(myChar)) {  // tests if myChar is a letter
  Serial.println("The character is a letter");
}
else {
  Serial.println("The character is not a letter");
}
```

isAlphaNumeric()
Description
Analyse if a char is alphanumeric (that is a letter or a numbers). Returns true if thisChar contains either a number or a letter.

Syntax
isAlphaNumeric(thisChar)

Parameters
thisChar: variable. Allowed data types: char.

Returns
true: if thisChar is alphanumeric.

Example Code
```
if (isAlphaNumeric(myChar)) { // tests if myChar isa letter or a number
  Serial.println("The character is alphanumeric");
}
else {
  Serial.println("The character is not alphanumeric");
}
```

isAscii()
Description
Analyse if a char is Ascii. Returns true if thisChar contains an Ascii character.

Syntax
isAscii(thisChar)

Parameters
thisChar: variable. Allowed data types: char.

Returns
true: if thisChar is Ascii.

Example Code
```
if (isAscii(myChar)) {  // tests if myChar is an Ascii character
  Serial.println("The character is Ascii");
}
else {
  Serial.println("The character is not Ascii");
}
```

isControl()
Description
Analyse if a char is a control character. Returns true if thisChar is a control character.

Syntax
isControl(thisChar)

Parameters
thisChar: variable. Allowed data types: char.

Returns
true: if thisChar is a control character.

Example Code
```
if (isControl(myChar)) {  // tests if myChar is a control character
  Serial.println("The character is a control character");
}
else {
  Serial.println("The character is not a control character");
}
```

isDigit()
Description
Analyse if a char is a digit (that is a number). Returns true if thisChar is a number.

Syntax
isDigit(thisChar)

Parameters
thisChar: variable. Allowed data types: char.

Returns
true: if thisChar is a number.

Example Code

```
if (isDigit(myChar)) {  // tests if myChar is a digit
  Serial.println("The character is a number");
}
else {
  Serial.println("The character is not a number");
}
```

isGraph()

Description

Analyse if a char is printable with some content (space is printable but has no content). Returns true if thisChar is printable.

Syntax

```
isGraph(thisChar)
```

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is printable.

Example Code

```
if (isGraph(myChar)) {  // tests if myChar is a printable character but not a blank space.
  Serial.println("The character is printable");
}
else {
  Serial.println("The character is not printable");
}
```

isHexadecimalDigit()

Description

Analyse if a char is an hexadecimal digit (A-F, 0-9). Returns true if thisChar contains an hexadecimal digit.

Syntax

```
isHexadecimalDigit(thisChar)
```

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is an hexadecimal digit.

Example Code

```
if (isHexadecimalDigit(myChar)) { // tests if myChar is an hexadecimal digit
  Serial.println("The character is an hexadecimal digit");
```

```
}
else {
  Serial.println("The character is not an hexadecimal digit");
}
```

isLowerCase()

Description

Analyse if a char is lower case (that is a letter in lower case). Returns true if thisChar contains a letter in lower case.

Syntax

isLowerCase(thisChar)

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is lower case.

Example Code

```
if (isLowerCase(myChar)) {  // tests if myChar is a lower case letter
  Serial.println("The character is lower case");
}
else {
  Serial.println("The character is not lower case");
}
```

isPrintable()

Description

Analyse if a char is printable (that is any character that produces an output, even a blank space). Returns true if thisChar is printable.

Syntax

isPrintable(thisChar)

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is printable.

Example Code

```
if (isPrintable(myChar)) {  // tests if myChar is printable char
  Serial.println("The character is printable");
}
else {
  Serial.println("The character is not printable");
}
```

isPunct()

Description

Analyse if a char is punctuation (that is a comma, a semicolon, an exlamation mark and so on). Returns true if thisChar is punctuation.

Syntax

isPunct(thisChar)

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is a punctuation.

Example Code

```
if (isPunct(myChar)) {  // tests if myChar is a punctuation character
  Serial.println("The character is a punctuation");
}
else {
  Serial.println("The character is not a punctuation");
}
```

isSpace()

Description

Analyse if a char is a white-space character. Returns true if the argument is a space, form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v').

Syntax

isSpace(thisChar)

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is a white-space character.

Example Code

```
if (isSpace(myChar)) {  // tests if myChar is a white-space character
  Serial.println("The character is white-space");
}
else {
  Serial.println("The character is not white-space");
}
```

isUpperCase()

Description

Analyse if a char is upper case (that is, a letter in upper case). Returns true if thisChar is upper case.

Syntax

isUpperCase(thisChar)

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is upper case.

Example Code

```
if (isUpperCase(myChar)) {  // tests if myChar is an upper case letter
  Serial.println("The character is upper case");
}
else {
  Serial.println("The character is not upper case");
}
```

isWhitespace()

Description

Analyse if a char is a space character. Returns true if the argument is a space or horizontal tab ('\t').

Syntax

isWhitespace(thisChar)

Parameters

thisChar: variable. Allowed data types: char.

Returns

true: if thisChar is a space character.

Example Code

```
if (isWhitespace(myChar)) { // tests if myChar is a space character
  Serial.println("The character is a space or tab");
}
else {
  Serial.println("The character is not a space or tab");
}
```

Random Numbers

- random()
- randomSeed()

random()

Description

The random function generates pseudo-random numbers.

Syntax

random(max)
random(min, max)

Parameters

min: lower bound of the random value, inclusive (optional).
max: upper bound of the random value, exclusive.

Returns

A random number between min and max-1. Data type: long.

Example Code

The code generates random numbers and displays them.

```
long randNumber;

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

  // if analog input pin 0 is unconnected, random analog noise will cause the call to randomSeed() to
//generate different seed numbers each time the sketch runs. randomSeed() will then shuffle the
//random function.
  randomSeed(analogRead(0));
}

void loop() {
  // print a random number from 0 to 299
  randNumber = random(300);
  Serial.println(randNumber);

  // print a random number from 10 to 19
  randNumber = random(10, 20);
  Serial.println(randNumber);

  delay(50);
}
```

randomSeed()

Description

randomSeed() initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence. This sequence, while very long, and random, is always the same.

Syntax

randomSeed(seed)

Parameters

seed: number to initialize the pseudo-random sequence. Allowed data types: unsigned long.

Returns

Nothing

Example Code

The code generates a pseudo-random number and sends the generated number to the serial port.

```
long randNumber;

void setup() {
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop() {
  randNumber = random(300);
  Serial.println(randNumber);
  delay(50);
}
```

Bits and Bytes

- bit()
- bitClear()
- bitRead()
- bitSet()
- bitWrite()
- highByte()
- lowByte()

bit()

Description

Computes the value of the specified bit (bit 0 is 1, bit 1 is 2, bit 2 is 4, etc.).

Syntax

bit(n)

Parameters n: the bit whose value to compute

Returns
The value of the bit.

bitClear()
Description
Clears (writes a 0 to) a bit of a numeric variable.

Syntax
bitClear(x, n)

Parameters
x: the numeric variable whose bit to clear.
n: which bit to clear, starting at 0 for the least-significant (rightmost) bit.

Returns
Nothing

bitRead()
Description
Reads a bit of a number.

Syntax
bitRead(x, n)

Parameters
x: the number from which to read.
n: which bit to read, starting at 0 for the least-significant (rightmost) bit.

Returns
The value of the bit (0 or 1).

bitSet()
Description
Sets (writes a 1 to) a bit of a numeric variable.

Syntax
bitSet(x, n)

Parameters
x: the numeric variable whose bit to set.
n: which bit to set, starting at 0 for the least-significant (rightmost) bit.

Returns
Nothing

bitWrite()

Description

Writes a bit of a numeric variable.

Syntax

bitWrite(x, n, b)

Parameters

x: the numeric variable to which to write.
n: which bit of the number to write, starting at 0 for the least-significant (rightmost) bit.
b: the value to write to the bit (0 or 1).

Returns

Nothing

Example Code

Demonstrates the use of bitWrite by printing the value of a variable to the Serial Monitor before and after the use of bitWrite().

```
void setup() {
  Serial.begin(9600);
  while (!Serial) {}  // wait for serial port to connect. Needed for native USB port only
  byte x = 0b10000000;  // the 0b prefix indicates a binary constant
  Serial.println(x, BIN); // 10000000
  bitWrite(x, 0, 1);  // write 1 to the least significant bit of x
  Serial.println(x, BIN); // 10000001
}

void loop() {
}
```

highByte()

Description

Extracts the high-order (leftmost) byte of a word (or the second lowest byte of a larger data type).

Syntax

highByte(x)

Parameters

x: a value of any type

Returns

Data type: byte.

lowByte()

Description

Extracts the low-order (rightmost) byte of a variable (e.g. a word).

Syntax
lowByte(x)

Parameters
x: a value of any type

Returns
Data type: byte.

External Interrupts

- attachInterrupt()
- detachInterrupt()

attachInterrupt()
Description
Digital Pins With Interrupts

The first parameter to attachInterrupt() is an interrupt number. Normally you should use digitalPinToInterrupt(pin) to translate the actual digital pin to the specific interrupt number. For example, if you connect to pin 3, use digitalPinToInterrupt(3) as the first parameter to attachInterrupt().

Syntax
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode) (recommended)
attachInterrupt(interrupt, ISR, mode) (not recommended)

Parameters
interrupt: the number of the interrupt. Allowed data types: int.
pin: the Arduino pin number.
ISR: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.
mode: defines when the interrupt should be triggered. Four constants are predefined as valid values:

- LOW to trigger the interrupt whenever the pin is low,
- CHANGE to trigger the interrupt whenever the pin changes value
- RISING to trigger when the pin goes from low to high,
- FALLING for when the pin goes from high to low.

Returns
Nothing

Example Code
```
const byte ledPin = 13;
const byte interruptPin = 2;
```

```
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

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

detachInterrupt()

Description

Turns off the given interrupt.

Syntax

detachInterrupt(digitalPinToInterrupt(pin)) (recommended)
detachInterrupt(interrupt) (not recommended)
detachInterrupt(pin) (Not recommended. Additionally, this syntax only works on Arduino SAMD Boards, Uno WiFi Rev2, Due, and 101.)

Parameters

interrupt: the number of the interrupt to disable (see attachInterrupt() for more details).
pin: the Arduino pin number of the interrupt to disable

Returns

Nothing

Interrupts

- interrupts()
- noInterrupts()

interrupts()

Description

Re-enables interrupts (after they've been disabled by nointerrupts(). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Syntax

interrupts()

Parameters
None

Returns
Nothing

Example Code
The code enables Interrupts.

```
void setup() {}

void loop() {
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

noInterrupts()
Description
Disables interrupts (you can re-enable them with interrupts()). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Syntax
noInterrupts()

Parameters
None

Returns
Nothing

Example Code
The code shows how to enable interrupts.

```
void setup() {}

void loop() {
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

Communication

- Serial
- Stream

Serial

Description

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART), and some have several.

You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to begin().

Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board). Don't connect these pins directly to an RS232 serial port; they operate at +/- 12V and can damage your Arduino board.

Stream

Description

Stream is the base class for character and binary based streams. It is not called directly, but invoked whenever you use a function that relies on it.

Stream defines the reading functions in Arduino. When using any core functionality that uses a read() or similar method, you can safely assume it calls on the Stream class. For functions like print(), Stream inherits from the Print class.

Some of the libraries that rely on Stream include :

- Serial
- Wire
- Ethernet
- SD

USB

- Keyboard
- Mouse

Keyboard

Description

The keyboard functions enable 32u4 or SAMD micro based boards to send keystrokes to an attached computer through their micro's native USB port.

When using the Mouse or Keyboard library, it may be best to test your output first using Serial.print(). This way, you can be sure you know what values are being reported.

Mouse

Description

The mouse functions enable 32u4 or SAMD micro based boards to control cursor movement on a connected computer through their micro's native USB port. When updating the cursor position, it is always relative to the cursor's previous location.