

A decorative graphic of a circuit board with various traces and components, rendered in a light blue color, framing the central text area.

PART – III

ARDUINO PROGRAMMING

THE STRUCTURE OF ARDUINO PROGRAM

- Preparation & Execution
- Each block has a set of statements enclosed in curly braces:

```
void setup( )
```

```
{
```

```
  statements-1;.
```

```
  .
```

```
  statement-n;
```

```
}
```

```
void loop ( )
```

```
{
```

```
  statement-1;.
```

```
  .
```

```
  statement-n;
```

```
}
```

Here, `setup ()` is the preparation block and `loop ()` is an execution block.

THE SET-UP

- The setup function is the first to execute when the program is executed, and this function is called only once.
- The setup function is used to initialize the pin modes and start serial communication.
- This function has to be included even if there are no statements to execute.
- After the setup () function is executed, the execution block runs next.

```
void setup ( )  
{  
    pinMode (pin-number, OUTPUT); // set the 'pin-  
number' as output  
    pinMode (pin-number, INPUT); // set the 'pin-number'  
as output  
}
```

EXECUTION BLOCK

- In the above example `loop ()` function is a part of execution block. As the name suggests, the `loop()` function executes the set of statements (enclosed in curly braces) repeatedly.

```
void loop ( )
```

```
{
```

```
  digitalWrite (pin-number, HIGH); // turns ON the  
  component connected to 'pin-number'
```

```
  delay (1000); // wait for 1 sec
```

```
  digitalWrite (pin-number, LOW); // turns OFF the  
  component connected to 'pin-number'
```

```
  delay (1000); //wait for 1sec
```

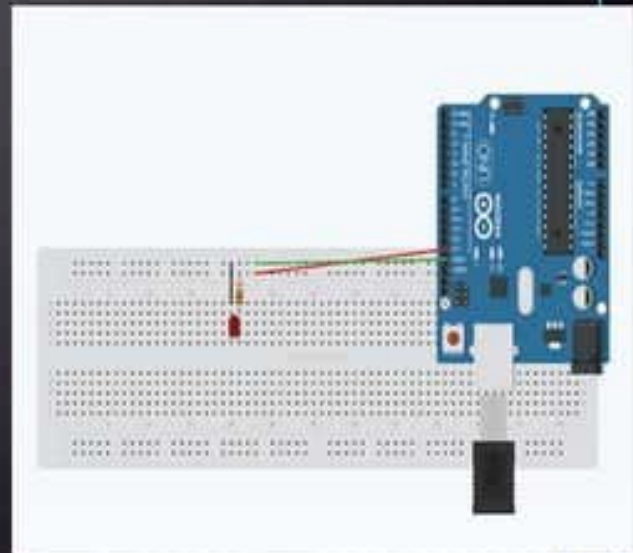
```
}
```

Note: Arduino always measures the time duration in millisecond.

Therefore, whenever you mention the delay, keep it in milli seconds.

EXAMPLE : LED BLINKING

- Steps in **BUILDING A BREADBOARD CONNECTION**:
- **Step-1**: Connect the Arduino to the Windows / Mac / Linux system via a USB cable
- **Step-2**: Connect the 13th digital pin of Arduino to the positive power rail of the breadboard and GND to the negative
- **Step-3**: Connect the positive power rail to the terminal strip via a 1K ohm resistor
- **Step-4**: Fix the LED to the ports below the resistor connection in the terminal strip
- **Step-5**: Close the circuit by connecting the cathode (the short chord) of the LED to the negative power strip of the breadboard



PROGRAM: LED BLINKING

```
void setup ( )  
{  
    pinMode (LED, OUTPUT); //Declaring pin 13 as output pin  
}  
void loop( ) // The loop function runs again and again  
{  
    digitalWrite (LED, HIGH); //Turn ON the LED  
    delay(1000); //Wait for 1sec  
    digitalWrite (LED, LOW); // Turn off the LED  
    delay(1000); // Wait for 1sec  
}
```

- 
- if statement
 - if else statement
 - for loop
 - while loop
 - do while loop
 - pinMode
 - digitalRead
 - digitalWrite
 - analogRead
 - analogWrite

IF STATEMENT

- **if statement**
- The “if” statement is a conditional statement, it checks if a certain condition is met. If yes, it executes the set of statements enclosed in curly braces. If the condition is false, then the set of statements will skip the execution.

The syntax of the “if” statement is follows:

```
if(some variable ?? state)
```

```
{  
  statement-1;  
  statement-n;  
}
```

Example

```
if (LED == HIGH)  
{  
  digitalWrite(LED, LOW);  
}
```

- In the syntax format, **??** represents **comparison operator**
- $X==Y$ // Check if X is equal to Y
- $X!=Y$ // X is not equal to Y
- $X<Y$ // Check if X is less than Y
- $X>Y$ // Check if X is greater than Y
- $X<=Y$ // Check if X is less than or equal to Y
- $X>=Y$ // Check if X is greater than or equal to Y
- Note: “=” is used to assign a value, where as == is used for comparison.

IF STATEMENT

- **if statement**
- The “if” statement is a conditional statement, it checks if a certain condition is met. If yes, it executes the set of statements enclosed in curly braces. If the condition is false, then the set of statements will skip the execution.

The syntax of the “if” statement is follows:

```
if(some variable ?? state)
```


```
{  
  statement-1;  
  statement-n;  
}
```

Example

```
if (LED == HIGH)  
{  
  digitalWrite(LED, LOW);  
}
```

- In the syntax format, **??** represents **comparison operator**
- $X==Y$ // Check if X is equal to Y
- $X!=Y$ // X is not equal to Y
- $X<Y$ // Check if X is less than Y
- $X>Y$ // Check if X is greater than Y
- $X<=Y$ // Check if X is less than or equal to Y
- $X>=Y$ // Check if X is greater than or equal to Y
- Note: “=” is used to assign a value, where as == is used for comparison.

IF-ELSE STATEMENT

- **if-else statement**
- This statement makes an "either-or" decision. The if statement checks a condition. If it is true, it executes a set of statements; if the condition is not true, it executes other set of statements.
- The syntax of the "if" statement is as : 

Example

```
if(LED==High)
{
  digitalWrite(LED, LOW);
}
else
{
  digitalWrite(LED, HIGH);
}
```

if (some variable ?? state)

{

statement-1;

statement-n;

}

else

{

statement-1;

statement-n;

}

FOR – WHILE LOOP

- for-loop
- If you want to repeatedly execute a set of statements for a specific number of times, then you can use a for loop.

for(initialization; condition; expression)

{

Statement-1;

Statement-n;

}

```
for (int X=0; X<50; X++)  
{  
  digitalWrite (12, HIGH); //Turn ON pin number 12  
  delay(1000);  
  digitalWrite ( 12, LOW); //Turn OFF pin number  
  12  
  delay (1000); //Wait for 1sec  
}
```

- while loop

- The while loop executes a set of statements until the expression inside the parentheses is false.

while (some variable ?? value)

{

Statement-1

Statement-n;

}

```
while(i<=200)  
{  
  digitalWrite (12, HIGH); //Turn on pin  
  number 12  
  delay(1000); //Wait for 1 sec  
  digitalWrite ( 12, LOW); //Turn pin number  
  12  
  delay (1000); //Wait for 1sec  
  i++;  
}
```

DO-WHILE LOOP

Do-while loop

- If you want to execute a set of statements once and repeatedly execute the set if a certain condition is true, the syntax of the “do-while” loop is as follows:

```
do
{
statement-1;
statement-n;
} while (some variable ?? value);
```

pinMode

- This statement is used in the preparation block of the Arduino program, that is, in the void setup() function.
- The pinMode statement is used to configure a pin to behave in either the INPUT mode or OUTPUT mode.
- The syntax of the “pinMode” statement is as follows:

```
pinMode (pin-number, behaviour);
```

PIN READ-WRITE COMMAND

digitalRead

This statement reads the state of the specified digital pin and the result will be either HIGH or LOW.

The **syntax** is as follows:

State=**digitalRead**(pin-number);

digitalWrite

digitalWrite statement is used to either turn ON or OFF the device connected to a specified digital pin.

The **syntax** of the “**digitalWrite**” statement is as follows:

digitalWrite(pin-number, status)

analogRead

This statement reads the value from the specified analog pin on the Arduino board with 10-bit resolution. The result will be an **integer value** in the **range of 0 to 1023**.

The **syntax** of the “**analogRead** statement” is as follows:

value= **analogRead** (pin-number);

analogWrite

This statement writes a pseudo-analog value to the specified pin. The value is called pseudo-analog because it is generated by Pulse Width Modulation pins (PWM) on the Arduino board. The **value** can be specified as a variable or constant in **the range of 0 to 255**.

The **syntax** of “**analogWrite**” statement is as follows:

analogWrite(pin-number, value);

loop()

After calling the `setup()` function, the `loop()` function does precisely what its name suggests, and loops consecutively, allowing the program to change, respond, and control the Arduino board.

```
void loop()
{
  digitalWrite(pin, HIGH); // turns 'pin' on  delay(1000);           // pauses for one second
  digitalWrite(pin, LOW);  // turns 'pin' off delay(1000);          // pauses for one second
}
```

structure | 7

functions

A function is a block of code that has a name and a block of statements that are executed when the function is called. The functions `void setup()` and `void loop()` have already been discussed and other built-in functions will be discussed later.

Custom functions can be written to perform repetitive tasks and reduce clutter in a program. Functions are declared by first declaring the function type. This is the type of value to be returned by the function such as 'int' for an integer type function. If no value is to be returned the function type would be void. After type, declare the name given to the function and in parenthesis any parameters being passed to the function.

```
type functionName(parameters)
{
  statements;
}
```

The following integer type function `delayVal()` is used to set a delay value in a program by reading the value of a potentiometer. It first declares a local variable `v`, sets `v` to the value of the potentiometer which gives a number between 0-1023, then divides that value by 4 for a final value between 0-255, and finally returns that value back to the main program.

```
int delayVal()
{
  int v;           // create temporary variable 'v'  v = analogRead(pot); // read
  potentiometer value  v /= 4;           // converts 0-1023 to 0-255  return v;           // return
  final value }
```

{ } curly braces

Curly braces (also referred to as just "braces" or "curly brackets") define the beginning and end of function blocks and statement blocks such as the void loop() function and the for and if statements.

```
type function()  
{ statements;  
}
```

An opening curly brace { must always be followed by a closing curly brace }. This is often referred to as the braces being balanced. Unbalanced braces can often lead to cryptic, impenetrable compiler errors that can sometimes be hard to track down in a large program.

The Arduino environment includes a convenient feature to check the balance of curly braces. Just select a brace, or even click the insertion point immediately following a brace, and its logical companion will be highlighted.

; semicolon

A semicolon must be used to end a statement and separate elements of the program. A semicolon is also used to separate elements in a for loop.

```
int x = 13; // declares variable 'x' as the integer 13
```

Note: Forgetting to end a line in a semicolon will result in a compiler error. The error text may be obvious, and refer to a missing semicolon, or it may not. If an impenetrable or seemingly illogical compiler error comes up, one of the first things to check is a missing semicolon, near the line where the compiler complained.

`/*... */` block comments

Block comments, or multi-line comments, are areas of text ignored by the program and are used for large text descriptions of code or comments that help others understand parts of the program. They begin with `/*` and end with `*/` and can span multiple lines.

```
/* this is an enclosed block comment    don't forget the closing  
comment -    they have to be balanced!  
*/
```

Because comments are ignored by the program and take no memory space they should be used generously and can also be used to “comment out” blocks of code for debugging purposes.

Note: While it is possible to enclose single line comments within a block comment, enclosing a second block comment is not allowed.

// line comments

Single line comments begin with `//` and end with the next line of code. Like block comments, they are ignored by the program and take no memory space.

```
// this is a single line comment
```

Single line comments are often used after a valid statement to provide more information about what the statement accomplishes or to provide a future reminder.

'inputVariable' is the variable itself. The first line declares that it will contain an int, short for integer. The second line sets the variable to the value at analog pin 2. This makes the value of pin 2 accessible elsewhere in the code.

Once a variable has been assigned, or re-assigned, you can test its value to see if it meets certain conditions, or you can use its value directly. As an example to illustrate three useful operations with variables, the following code tests whether the inputVariable is less than 100, if true it assigns the value 100 to inputVariable, and then sets a delay based on inputVariable which is now a minimum of 100:

```
if (inputVariable < 100) // tests variable if less than 100
{
  inputVariable = 100; // if true assigns value of 100
}
delay(inputVariable); // uses variable as delay
```

Note: Variables should be given descriptive names, to make the code more readable. Variable names like tiltSensor or pushButton help the programmer and anyone else reading the code to understand what the variable represents. Variable names like var or value, on the other hand, do little to make the code readable and are only used here as examples. A variable can be named any word that is not already one of the keywords in the Arduino language.

variable declaration

All variables have to be declared before they can be used. Declaring a variable means defining its value type, as in int, long, float, etc., setting a specified name, and optionally assigning an initial value. This only needs to be done once in a program but the value can be changed at any time using arithmetic and various assignments.

The following example declares that inputVariable is an int, or integer type, and that its initial value equals zero. This is called a simple assignment.

```
int inputVariable = 0;
```

A variable can be declared in a number of locations throughout the program and where this definition takes place determines what parts of the program can use the variable.

variable scope

A variable can be declared at the beginning of the program before `void setup()`, locally inside of functions, and sometimes within a statement block such as for loops. Where the variable is declared determines the variable scope, or the ability of certain parts of a program to make use of the variable.

A global variable is one that can be seen and used by every function and statement in a program. This variable is declared at the beginning of the program, before the `setup()` function.

A local variable is one that is defined inside a function or as part of a for loop. It is only visible and can only be used inside the function in which it was declared. It is therefore possible to have two or more variables of the same name in different parts of the same program that contain different values. Ensuring that only one function has access to its variables simplifies the program and reduces the potential for programming errors.

The following example shows how to declare a few different types of variables and demonstrates each variable's visibility:

```
int value;          // 'value' is visible          // to any function void
setup()
{
  // no setup needed
}
void loop()
{
  for (int i=0; i<20; i++) // 'i' is only visible {          // inside the for-loop
  {
    i++;
  }
  float f;          // 'f' is only visible
}                  // inside loop
```

byte

Byte stores an 8-bit numerical value without decimal points. They have a range of 0255.

```
byte someVariable = 180; // declares 'someVariable'           //  
as a byte type
```

int

Integers are the primary datatype for storage of numbers without decimal points and store a 16-bit value with a range of 32,767 to -32,768.

```
int someVariable = 1500; // declares 'someVariable'  
                        // as an integer type
```

Note: Integer variables will roll over if forced past their maximum or minimum values by an assignment or comparison. For example, if $x = 32767$ and a subsequent statement adds 1 to x , $x = x + 1$ or $x++$, x will then rollover and equal -32,768.

long

Extended size datatype for long integers, without decimal points, stored in a 32-bit value with a range of 2,147,483,647 to -2,147,483,648.

```
long someVariable = 90000; // declares 'someVariable'  
                        // as a long type
```

float

A datatype for floating-point numbers, or numbers that have a decimal point. Floatingpoint numbers have greater resolution than integers and are stored as a 32-bit value with a range of 3.4028235E+38 to -3.4028235E+38.

```
float someVariable = 3.14; // declares 'someVariable'           // as a floating-  
point type
```

Note: Floating-point numbers are not exact, and may yield strange results when compared. Floating point math is also much slower than integer math in performing calculations, so should be avoided if possible.

arrays

An array is a collection of values that are accessed with an index number. Any value in the array may be called upon by calling the name of the array and the index number of the value. Arrays are zero indexed, with the first value in the array beginning at index number 0. An array needs to be declared and optionally assigned values before they can be used.

```
int myArray[] = {value0, value1, value2...}
```

Likewise it is possible to declare an array by declaring the array type and size and later assign values to an index position:

```
int myArray[5]; // declares integer array w/ 6 positions myArray[3] = 10; // assigns the 4th index the value 10
```

To retrieve a value from an array, assign a variable to the array and index position:

```
x = myArray[3]; // x now equals 10
```

Arrays are often used in for loops, where the increment counter is also used as the index position for each array value. The following example uses an array to flicker an LED. Using a for loop, the counter begins at 0, writes the value contained at index position 0 in the array flicker[], in this case 180, to the PWM pin 10, pauses for 200ms, then moves to the next index position.

```
int ledPin = 10;           // LED on pin 10 byte flicker[] = {180, 30, 255, 200, 10, 90, 150, 60};  
                           // above array of 8 void setup()           // different values  
{  
  pinMode(ledPin, OUTPUT); // sets OUTPUT pin  
}  
void loop()  
{  
  for(int i=0; i<7; i++)    // loop equals number {           // of values in array  
    analogWrite(ledPin, flicker[i]); // write index value  delay(200);           // pause 200ms  
  }  
}
```

arithmetic

Arithmetic operators include addition, subtraction, multiplication, and division. They return the sum, difference, product, or quotient (respectively) of two operands.

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

The operation is conducted using the data type of the operands, so, for example, $9 / 4$ results in 2 instead of 2.25 since 9 and 4 are ints and are incapable of using decimal points. This also means that the operation can overflow if the result is larger than what can be stored in the data type.

If the operands are of different types, the larger type is used for the calculation. For example, if one of the numbers (operands) are of the type float and the other of type integer, floating point math will be used for the calculation.

Choose variable sizes that are large enough to hold the largest results from your calculations. Know at what point your variable will rollover and also what happens in the other direction e.g. (0 - 1) OR (0 - - 32768). For math that requires fractions, use float variables, but be aware of their drawbacks: large size and slow computation speeds.

Note: Use the cast operator e.g. (int)myFloat to convert one variable type to another on the fly. For example, `i = (int)3.6` will set i equal to 3.

compound assignments

Compound assignments combine an arithmetic operation with a variable assignment. These are commonly found in for loops as described later. The most common compound assignments include:

`x ++` // same as `x = x + 1`, or increments `x` by `+1` `x --` // same as `x = x - 1`, or decrements `x` by `-1` `x += y` // same as `x = x + y`, or increments `x` by `+y` `x -= y` // same as `x = x - y`, or decrements `x` by `-y` `x *= y` // same as `x = x * y`, or multiplies `x` by `y` `x /= y` // same as `x = x / y`, or divides `x` by `y`

Note: For example, `x *= 3` would triple the old value of `x` and re-assign the resulting value to `x`.

comparison operators

Comparisons of one variable or constant against another are often used in if statements to test if a specified condition is true. In the examples found on the following pages, ?? is used to indicate any of the following conditions:

`x == y` // x is equal to y
`x != y` // x is not equal to y
`x < y` // x is less than y
`x > y` // x is greater than y
`x <= y` // x is less than or equal to y
`x >= y` // x is greater than or equal to y

logical operators

Logical operators are usually a way to compare two expressions and return a TRUE or FALSE depending on the operator. There are three logical operators, AND, OR, and NOT, that are often used in if statements:

Logical AND:

`if (x > 0 && x < 5)` // true only if both // expressions are true

Logical OR:

`if (x > 0 || y > 0)` // true if either // expression is true

Logical NOT:

`if (!x > 0)` // true only if // expression is false

constants

The Arduino language has a few predefined values, which are called constants. They are used to make the programs easier to read. Constants are classified in groups.

true/false

These are Boolean constants that define logic levels. FALSE is easily defined as 0 (zero) while TRUE is often defined as 1, but can also be anything else except zero. So in a Boolean sense, -1, 2, and -200 are all also defined as TRUE.

```
if (b == TRUE);  
{  
  doSomething;  
}
```

high/low

These constants define pin levels as HIGH or LOW and are used when reading or writing to digital pins. HIGH is defined as logic level 1, ON, or 5 volts while LOW is logic level 0, OFF, or 0 volts.

```
digitalWrite(13, HIGH);
```

input/output

Constants used with the pinMode() function to define the mode of a digital pin as either INPUT or OUTPUT.

```
pinMode(13, OUTPUT);
```


if

if statements test whether a certain condition has been reached, such as an analog value being above a certain number, and executes any statements inside the brackets if the statement is true. If false the program skips over the statement. The format for an if test is:

```
if (someVariable ?? value)
{
    doSomething;
}
```

The above example compares someVariable to another value, which can be either a variable or constant. If the comparison, or condition in parentheses is true, the statements inside the brackets are run. If not, the program skips over them and continues on after the brackets.

Note: Beware of accidentally using '=', as in if(x=10), while technically valid, defines the variable x to the value of 10 and is as a result always true. Instead use '==', as in if(x==10), which only tests whether x happens to equal the value 10 or not. Think of '=' as "*equals*" opposed to '==' being "*is equal to*".

if... else

if... else allows for 'either-or' decisions to be made. For example, if you wanted to test a digital input, and do one thing if the input went HIGH or instead do another thing if the input was LOW, you would write that this way:

```
if (inputPin == HIGH)
{
    doThingA;
} else {
    doThingB;
}
```

else can also precede another if test, so that multiple, mutually exclusive tests can be run at the same time. It is even possible to have an unlimited number of these else branches. Remember though, only one set of statements will be run depending on the condition tests:

```
if (inputPin < 500)
{
    doThingA;
}
else if (inputPin >= 1000)
{
    doThingB;
} else {
    doThingC;
}
```

Note: An if statement simply tests whether the condition inside the parenthesis is true or false. This statement can be any valid C statement as in the first example, if (inputPin == HIGH). In this example, the if statement only checks to see if indeed the specified input is at logic level high, or +5v.

for

The for statement is used to repeat a block of statements enclosed in curly braces a specified number of times. An increment counter is often used to increment and terminate the loop. There are three parts, separated by semicolons (;), to the for loop header:

```
for (initialization; condition; expression)
{
    doSomething;
}
```

The initialization of a local variable, or increment counter, happens first and only once. Each time through the loop, the following condition is tested. If the condition remains true, the following statements and expression are executed and the condition is tested again. When the condition becomes false, the loop ends.

The following example starts the integer *i* at 0, tests to see if *i* is still less than 20 and if true, increments *i* by 1 and executes the enclosed statements:

```
for (int i=0; i<20; i++) // declares i, tests if less {           // than 20, increments i by 1
digitalWrite(13, HIGH); // turns pin 13 on   delay(250);       // pauses for 1/4 second
digitalWrite(13, LOW);  // turns pin 13 off  delay(250);       // pauses for 1/4 second
}
```

Note: The C for loop is much more flexible than for loops found in some other computer languages, including BASIC. Any or all of the three header elements may be omitted, although the semicolons are required. Also the statements for initialization, condition, and expression can be any valid C statements with unrelated variables. These types of unusual for statements may provide solutions to some rare programming problems.

while

while loops 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.

```
while (someVariable ?? value)
{
    doSomething;
}
```

The following example tests whether 'someVariable' is less than 200 and if true executes the statements inside the brackets and will continue looping until 'someVariable' is no longer less than 200.

```
while (someVariable < 200) // tests if less than 200
{
    doSomething;           // executes enclosed statements
    someVariable++;         // increments
    variable by 1 }        variable by 1 }
```

do... while

The do loop is a bottom driven loop that 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.

```
do {  
    doSomething;  
} while (someVariable ?? value);
```

The following example assigns readSensors() to the variable 'x', pauses for 50 milliseconds, then loops indefinitely until 'x' is no longer less than 100:

```
do {  
    x = readSensors(); // assigns the value of          //  
    readSensors() to x delay (50); // pauses 50 milliseconds } while  
(x < 100); // loops if x is less than 100
```

pinMode(pin, mode)

Used in void setup() to configure a specified pin to behave either as an INPUT or an OUTPUT.

```
pinMode(pin, OUTPUT); // sets 'pin' to output
```

Arduino digital pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode(). Pins configured as INPUT are said to be in a high-impedance state.

There are also convenient 20K Ω pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed in the following manner:

```
pinMode(pin, INPUT); // set 'pin' to input digitalWrite(pin, HIGH); // turn on pullup resistors
```

Pullup resistors would normally be used for connecting inputs like switches. Notice in the above example it does not convert pin to an output, it is merely a method for activating the internal pull-ups.

Pins configured as OUTPUT are said to be in a low-impedance state and can provide 40 mA (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (don't forget the series resistor), but not enough current to run most relays, solenoids, or motors.

Short circuits on Arduino pins and excessive current can damage or destroy the output pin, or damage the entire Atmega chip. It is often a good idea to connect an OUTPUT pin to an external device in series with a 470 Ω or 1K Ω resistor.

digitalRead(pin)

Reads the value from a specified digital pin with the result either HIGH or LOW. The pin can be specified as either a variable or constant (0-13).

```
value = digitalRead(Pin); // sets 'value' equal to
                        // the input pin
```

digitalWrite(pin, value)

Outputs either logic level HIGH or LOW at (turns on or off) a specified digital pin. The pin can be specified as either a variable or constant (0-13).

```
digitalWrite(pin, HIGH); // sets 'pin' to high
```

The following example reads a pushbutton connected to a digital input and turns on an LED connected to a digital output when the button has been pressed:

```
int led = 13; // connect LED to pin 13 int pin = 7; // connect pushbutton to pin 7
int value = 0; // variable to store the read value

void setup()
{
  pinMode(led, OUTPUT); // sets pin 13 as output  pinMode(pin, INPUT); // sets pin 7 as input
}
void loop()
{
  value = digitalRead(pin); // sets 'value' equal to // the input pin
  digitalWrite(led, value); // sets 'led' to the
} // button's value
```


analogRead(pin)

Reads the value from a specified analog pin with a 10-bit resolution. This function only works on the analog in pins (0-5). The resulting integer values range from 0 to 1023.

```
value = analogRead(pin); // sets 'value' equal to 'pin'
```

Note: Analog pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

analogWrite(pin, value)

Writes a pseudo-analog value using hardware enabled pulse width modulation (PWM) to an output pin marked PWM. On newer Arduinos with the ATmega168 chip, this function works on pins 3, 5, 6, 9, 10, and 11. Older Arduinos with an ATmega8 only support pins 9, 10, and 11. The value can be specified as a variable or constant with a value from 0-255.

```
analogWrite(pin, value); // writes 'value' to analog 'pin'
```

A value of 0 generates a steady 0 volts output at the specified pin; a value of 255 generates a steady 5 volts output at the specified pin. For values in between 0 and 255, the pin rapidly alternates between 0 and 5 volts - the higher the value, the more often the pin is HIGH (5 volts). For example, a value of 64 will be 0 volts threequarters of the time, and 5 volts one quarter of the time; a value of 128 will be at 0 half the time and 255 half the time; and a value of 192 will be 0 volts one quarter of the time and 5 volts three-quarters of the time.

Because this is a hardware function, the pin will generate a steady wave after a call to analogWrite in the background until the next call to analogWrite (or a call to digitalWrite or digitalWrite on the same pin).

Note: Analog pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

The following example reads an analog value from an analog input pin, converts the value by dividing by 4, and outputs a PWM signal on a PWM pin:

```
int led = 10; // LED with 220 resistor on pin 10 int pin = 0; // potentiometer on
analog pin 0 int value; // value for reading

void setup(){ // no setup needed

void loop()
{
  value = analogRead(pin); // sets 'value' equal to 'pin' value /= 4; // converts 0-
1023 to 0-255 analogWrite(led, value); // outputs PWM signal to led }
```

delay(ms)

Pauses a program for the amount of time as specified in milliseconds, where 1000 equals 1 second.

```
delay(1000); // waits for one second
```

millis()

Returns the number of milliseconds since the Arduino board began running the current program as an unsigned long value.

```
value = millis(); // sets 'value' equal to millis()
```

Note: This number will overflow (reset back to zero), after approximately 9 hours.

min(x, y)

Calculates the minimum of two numbers of any data type and returns the smaller number.

```
value = min(value, 100); // sets 'value' to the smaller of  
                        // 'value' or 100, ensuring that           // it never gets above 100.
```

max(x, y)

Calculates the maximum of two numbers of any data type and returns the larger number.

```
value = max(value, 100); // sets 'value' to the larger of  
                        // 'value' or 100, ensuring that           // it is at least 100.
```